Crypto Chips

SODAQ

Introduction

Simply put, crypto chips, also known as cryptographic co-processors and crypto-authentication chips, are application-specific processors that hold one or more keys, and uses these in the encryption of data. This allows the user to truly encrypt anything end-to-end.

Such a crypto chip is 'just' another chip on a PCB. Depending on the interfaces and functionality provided, a standard IC size is selected. A common size is an 8-pin package: these are no larger than 6 x 5 x 2 mm.

Why we need Security

In an ideal world we would not need security and privacy, since it is granted throughout. but since we live in a non-ideal world, we do.

The need for security and security measures are a byproduct of the existence of individuals or organizations with malicious intent. When it comes to Internet of Things and LoRaWAN, some examples are device impersonation, overwriting device firmware, and software, and data piracy.

If someone wishes to listen to the data output of your device, they simply need a short period of physical access to the device to identify the session keys, if even that. The data transmitted by the board is then also received by them, and can be used to gain some advantage: think about armoured value-transporters: attaching a global positioning device, to be transmitted to a control center, to identify issues with the truck (hijacking, or engine trouble). Just receiving this position is sufficient for thieves to know when and where to strike. In such scenarios, it is imperative that any data transmitted is (securely) encrypted to prevent misuse.

So, humanity, choose one: encrypt your data, or get your act together.

Capabilities of a Crypto Chip

The most commonly thought-of purpose of crypto chips is the encryption of (outgoing) data. At the receiving end, a crypto chip can be used to efficiently decrypt the incoming message. The message is encrypted using either symmetric encryption, or asymmetric encryption.

Another commonly used function is the storage of security keys. This happens in a secure element inside the device. The name says it: a secure element is a piece of hardware that is, in the practical sense, an impenetrable bank vault for bits and bytes. The keys stored here are those that are only supposed to be used internally; never leaving the crypto chip.

An example of these keys are root keys or certificates: these are some special signatures that are implemented in software or firmware before it is uploaded to the device. The device bootloader can then ask the crypto chip to compute the key and compare it to its list of root keys - if it doesn't match, the software is rejected.

Why do you need a Crypto Chip?

As previously explained, if data is critical for some operation, security through encryption becomes relevant. This is especially true, when that operation is in some way critical or proprietary, where knowledge about some of its parameters may compromise certain aspects of, or the entire operation. This means that the secrecy of that data is imperative to the sustainability of the operation, and the communication channels should not be trusted with the encryption parameters: this is where a crypto chip becomes essential.

In addition to data protection, software authentication is very useful, as previously mentioned. This ensures that only the owner of the device can overwrite firmware and software loaded to the device.

Relevance to LoRaWAN

LoRaWAN adds two symmetric layers of encryption on top of the payload. The application session key is used for the first layer, and the network session key is used for the second. The receiving network server then decrypts the top layer of incoming data set and pushes the remaining data set to the application server. The application server has the last session key used to decrypt the last layer, and can access the payload.

However, common use cases show that the network server is usually used to decrypt both of these layers, hence only one layer of encryption would have been sufficient: since two devices, of which one, the server, is not an end-device, know both sets of encryption keys, and plain data is then transmitted to the true application server. Even if this plain information is neglected, the network server is more open to attacks to reveal sets of encryption keys.

Then, three different scenarios arise: plain data is sniffed between the network server and application server; the network and application session keys being commonly known; or even the network server owner using your data for their own benefit. In those scenarios, a crypto chip, preferably working with an asymmetric encryption algorithm, can make the data secure if all else fails.

How to integrate Crypto Chips

Whereas crypto chips are very complex to work with due to their cryptographic origin, the devices are very much widely available at major electronic components retailers.

Depending on the device specifics, communication usually occurs between a microprocessor and the crypto chip on on of I^2C, SPI, or one-wire interfaces, and communication is usually specified in a freely accessible datasheet.

Theoretical Background

We can start discussing security, crypto chips and the implementation of crypto chips, but those topics do not become relevant until one has understood what they are protecting, and who, but also what they are protecting it from.

In the modern data age, attacks can be performed on Internet of Things devices, and computers in general. These attacks can be both physical or digital. Where physical access to the system implies that the attacker has physical access to monitor data channels, such as UART or I^2C communications. Due to the unencrypted nature of on-board communications, physical access can quickly qualify as a breach of the system in most cases, a portion may be less accessible (or visible), and thus likely less susceptible to the attacker's first attempts to get access. In the case of microprocessors and most Internet of Things devices, there is no operating system installed. Thus the attacker cannot gain access to a command shell, since it does not exist; but the on-board communication can be used to gain the transmission encryption keys. Another approach is to copy the installed program binary, overwriting the software or firmware to simply request the encryption keys desired, followed by reloading the program binary.

Taking the digital approach, the attacker requires some form of connection to the system. The attacker must then identify some weakness in the software to exploit, which can be barricaded almost entirely if the device is programmed properly and bug-free and extensively tested against potential software exploit approaches. The more complex the system becomes, the less likely this perfect implementation becomes.

Another form, that is more common in Internet of Things applications, is the interception of data, often referred to as packet sniffing. If or once the data is unencrypted, it can be considered an intellectual loss. Since these devices operate on internet-based communication, any of such communication should thus have some form of end-to-end encryption, with the possibility of additional encryption layers, in order to prevent a leak of usable data. This is, significantly more difficult to decrypt (or derive security keys from), and will be a lower priority for an attacker (attempting to retrieve the security keys from one of the two ends is an easier target than a constantly changing set of data).

But let's not forget that nothing is perfect: a brute force attack, if given sufficient time, will always succeed.

Encryption Types

There are two general types of encryption, being symmetric encryption and asymmetric (also known as public key) encryption.

In symmetric encryption, both the transmitting and receiving parties or devices have the same key, which is used for encryption, and for decryption. The flaw in this encryption method is that both parties know the key, and if either is uncovered, the messages are no longer secure. This encryption type requires the encryption transformation to be one-to-one (and thus reversible).

Contrarily, in asymmetric encryption, a publicly known key is used for encryption, and only the receiving party knows the private key that can be used for decryption, whereas the public key cannot be used to decrypt the message. In this case, the encryption is not reversible using the public key. The public key must thus be determined based upon some non-linear algorithm applied to the private key.

A variation on the asymmetric encryption is the Diffie-Hellman key exchange (a previously patented technique, invented by Diffie, Hellman and Merkle), where a set of parameters are publicly known. Using this set of parameters, and for both end-devices a private parameter (the private key), a non-linear calculation is performed, arriving at another number that is then made public: the public key. Since the mathematical operations include powers and modulus, the public key is a parameter that cannot be used to calculate the private key, as there is an infinite set private keys that could have been used to arrive at the public key. Using the other party's public key, and the own private key, in

both cases, allows the computation of a shared secret, which is in turn used for the encryption of the message.

Crypto Chips in LoRaWAN

Why should you use a crypto chip in a LoRaWAN device? The answer is simple: to ensure that the data transfer is secure end-to-end. But how would this work?

At the very least, the device and its application (the entire chain from device to application server) should be functional before the crypto chip is added, otherwise it adds no value. The crypto chip should, in the most basic of implementations, be used for encrypting the data to be communicated following an encryption protocol, using a key that can be used for decryption at the receiving end (symmetric encryption).

In such an application, the central processor retrieves data from sensors, performs some cleanup or conversion operation, so that the data can be sent in a smaller data set. The processor then sends the data to the crypto chip, with the instruction to encrypt using a key or set of keys preprogrammed in the crypto chip. The crypto chip encrypts and returns the encrypted data, which is then stored or passed on to the LoRa module for transmission. At the receiving end, the encryption protocol and encryption key must be known, in order to decrypt the data for evaluation. A similar crypto chip may be dedicated for this task.

At the hardware level of the device, the path of the data is illustrated in Figure 1:

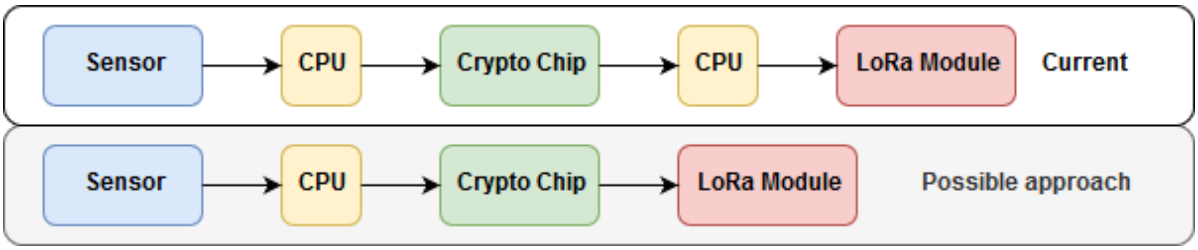


Figure 1: The most commonly used data path on-board, with a possible approach that limits encrypted data travel on the chip.

Encryption and Session Keys

The crypto chips also contain a secure element; a data vault for a limited set of encryption keys. The registers in this secure element are generally write only, or even prohibited for any external interaction; as the ability to read the key registers means that physical access to the device is a guaranteed vulnerability. This means that even the owner of the device cannot retrieve the keys. Some registers, however, may be readable, where the secure element holds a key for later retrieval and use: this allows the operator to store and, if required, retrieve the LoRa session keys, for example, but also allows making changes to firmware for a series of devices, rather than tailoring a specific set of lines in the code for each individual device.

The keys stored in the secure element registers can generally be used for the symmetric and asymmetric encryption of data, depending on the application preferences. Another set of keys that can be added to the registers, are root keys. These keys can be used to compare an answer for validity. This answer is computed by either the crypto chip or bootloader over the entire software that is intended to be uploaded. If it does not match, the bootloader can then reject the code, and may have the LoRa module transmit an error message, notifying the owner.

When to add Asymmetric Encryption?

Asymmetric encryption becomes especially useful when an operator requires a large number of replaceable devices transmitting information to a server, and wishes to have one master decryption process running, using only one key (as this is simpler to implement than a custom key per device).

Using asymmetric encryption then allows the server to periodically broadcast the derived public key, such that all the devices can encrypt using that public key. This is also useful if the decryption key is thought cracked, at which point the server can generate a new private key, and derive its public counterpart to transmit to the devices.

Since this requires cracking one key only, it may be worthwhile to explore more complicated encryption types (Diffie-Hellman, for example), in addition to adding separate key pairs per device. This becomes vastly more difficult for an unbiased brute force attack, as the time to crack scales per device added. This would prevent cracking attempts, but rather forces an attacker to search for other weaknesses to target.

Secure Applications

Ideally, a purely secure application is only vulnerable to brute force attacks; the most computationally intensive attacks, since it goes through all possibilities, starts at protected hardware, communicates encrypted data over a protected line with a protected protocol, to protected hardware.

Before encrypting any data, let's see how this ideally secure application compares to a typical LoRaWAN application. We know that the devices on both ends must be secure, but since it is entirely dependent on you or your hardware and software engineers, we will not evaluate that. The LoRa physical layer itself does not add encryption, but does encode the data into a CHIRP scheme, while easy to decode, it does add a boundary if unknown. LoRaWAN adds the Wide Area Network interfacing protocols between the receiving gateway and network server. As known, it adds two sets of symmetric encryption on top of the payload, to prevent eavesdropping of plain data before reaching the application server. While the line until the application server is not physically secure, the data transmission is.

Since in most scenarios the network server and application server are implemented together, the data is transmitted as plain data from the application server to the true application server. Preventing eavesdropping on this line is then the issue to address with another set of encryption.

When a variant of asymmetric encryption is added to the data before it is passed to the LoRaWAN module, performed by a crypto chip with a key in it; safe at hardware-level, the data is in instance protected from origin device to final destination.

The general communication and encryption structure then becomes similar to Figure 2:

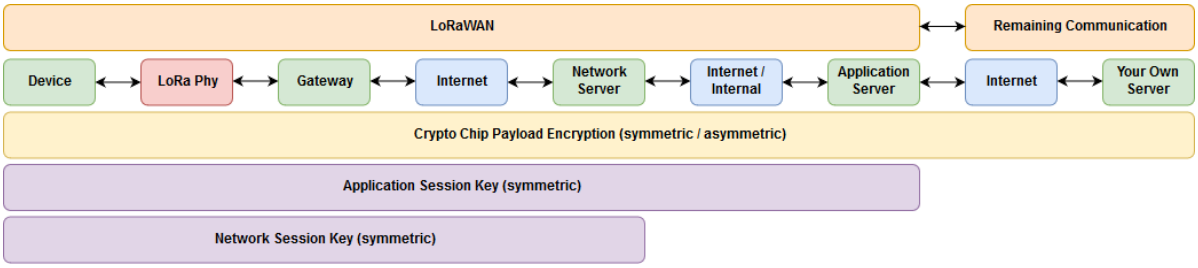


Figure 2: Block-diagram of a secure LoRaWAN application with encryption layers indicated.

But what if we want to make it even more difficult?

Beyond plain and encrypted data, even plain data means nothing if you don't know what each bit represents. The efficient data formats used take care of this to some degree. It is possible to go further and apply a standard (internally defined) scramble: bytes can be scrambled before or after encryption, or even both. Just ensure the application server knows how to reverse this scramble.

It doesn't end there! The data itself can be shifted for an even lower payload size, which then requires the eavesdropper to not only decrypt the set of data, know the order of scrambling, but also requires knowledge about data shifts and conversions before they are able to interpret the data.

ExpLoRer Crypto Chip Tutorial

The SODAQ ExpLoRer is quite unique with its on-board battery, battery charger circuit, LoRa and Bluetooth LE modules, temperature sensor and SPI flash, but especially the inclusion of a crypto chip: the Microchip ATECC508a (on boards before ExpLoRer revision 6c) or the Microchip ATECC608a; the latter being the later generation of the chip, and is the same apart from some additional features. Most notably, a hardware AES encryption feature has been added.

The chip is located on the ExpLoRer board as indicated in Figure 3:

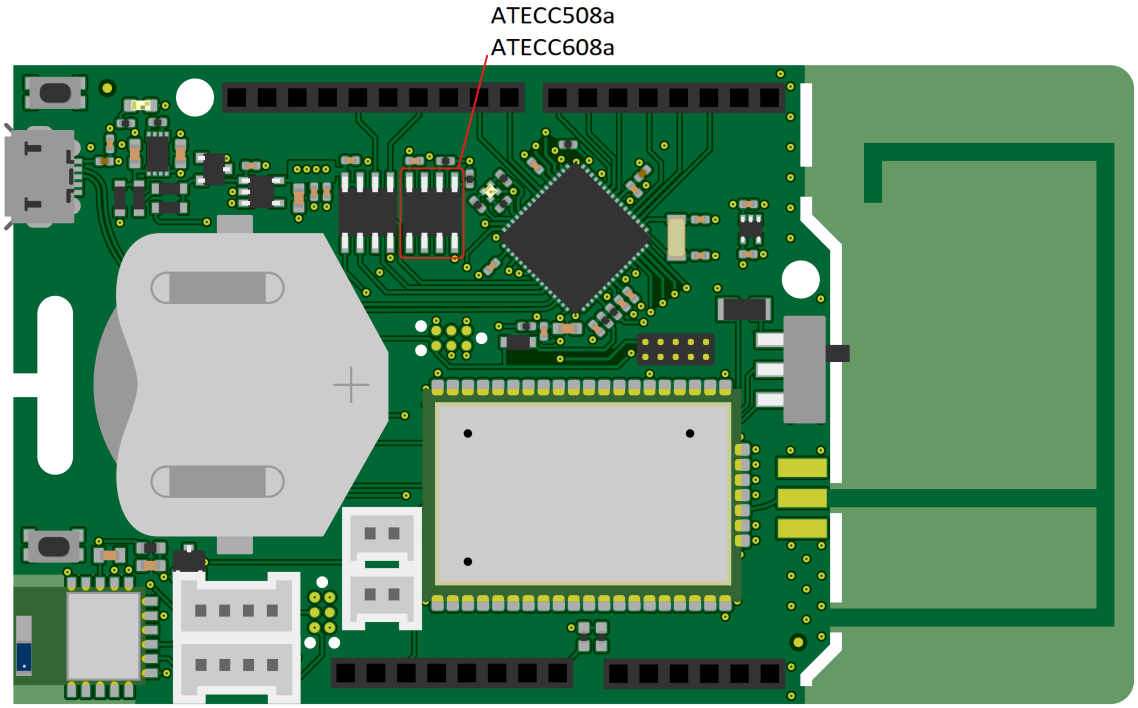


Figure 3: Crypto Chip location on the ExpLoRer board.

Please do understand that this tutorial is intended to help with some understanding, but in no way encompasses the complete instruction set; refer to the manufacturer, Microchip, and the chip-specific data sheet for more details.

A brief Description of the ATECC608a

As the name indicates, the ATECC chips are designed for and most effectively used for ECC, or Elliptic Curve Cryptography, a type of asymmetric cryptography. ECC is generally known for smaller key sizes for the same decryption difficulty, when compared to other cryptography schemes. The ATECC chip has a number of features, most notably a configurable secure data storage (secure element), and a true random number generator. The chip operates on the I^2C bus with high clock speeds: up to 1 MHz!

WARNING - A Piece of Advice

Before we get to the functionality of the chip, it is imperative that you understand that the function and security of the ATECC chips is guaranteed, but only by **permanently locking the configuration and data zones!** It is therefore urgently recommended that instead of locking the chip on the ExpLoRer,

the CryptoAuthentication Starter Kit[1] is purchased, which allows solderless interchanging of low-cost SMD chips (and includes a few different crypto chips to start with). This kit interfaces with the ACES Crypto Evaluation studio. A Microchip webinar provides some basics for using this software[2].

It is not advised to replace the ATECC chip on the ExpLoRer, due to the potential damage to nearby chips or the remainder of the board. If you do choose to do this, it's suggested to remove the coin cell battery and nearby plastic: the headers, for example.

Zone Locks

There are two degrees of locking. The first is the configuration zone lock: this is required before even the Random Number Generator (RNG) provides a random output. It is essential that you configure the chip's data slots to the required settings prior to enabling this lock; the chip needs to be replaced by a new one if the configuration was not set correctly.

The second lock additionally locks the data zone and One Time Programmable zone (not used in this tutorial), where the data zone slots inherit the WriteConfig and ReadConfig configuration settings that were locked-in by the first lock.

Message structure

Although the library will take care of most of the specifics named here, it is useful to know the following pieces of information.

The ATECC chips expect a message consisting of a leading Count byte, detailing the total transmission length in bytes, including itself (the Count byte) and two trailing checksum bytes.

All payload messages up to 2 bytes, configuration values and message headers should be ordered in Little Endian byte order. This can especially confusing, since the data sheet orders the numbers Big Endian for easier interpretation and only mentions this once.

Conversely, a payload, such as a key, should be transmitted in Big Endian order.

Finally, the transmission is ended with a two-byte checksum, again transmitted in Little Endian order. The checksum is calculated according to the CRC-16 algorithm, using the CRC polynomial $0x8005$.

ATECC608a Zone Specifics

As previously described, the ATECC consists of multiple zones, each individually configurable. Following are some descriptions of what can be configured, and where applicable, an example configuration. To aid the explanation, some screenshots of the Microchip ACES Crypto Evaluation studio are included.

Configuration Zone

The Configuration Zone can always be read by external operators such as the MCU, though the nature of the response, encrypted or plain text, is dependent on the configuration of the zone itself. Naturally, it can be written to prior to the Configuration Lock, but not afterwards.

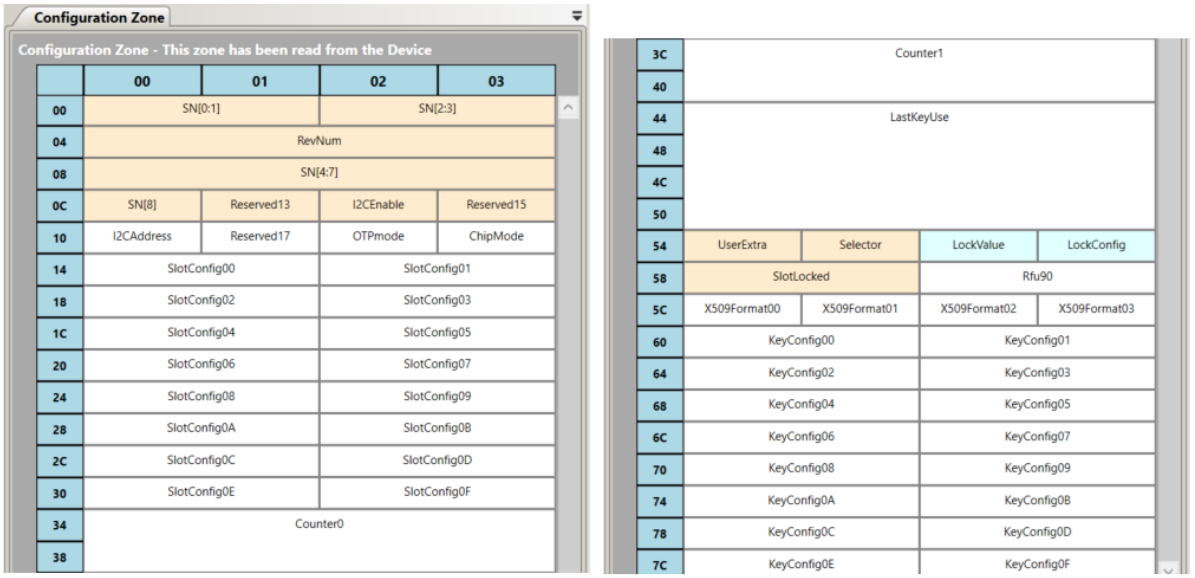
The Configuration Zone consists of 128 bytes and, as presented in ACES, is shown in Figure 4:

Where the indices of the configuration bytes are noted on the sides. The white cells are writable, the blue cells, the lock parameters, are writable as well, but before adaption, the written value is put through a bit-wise AND comparator on the current contents of the register, in this case the LOCK parameters (for example, if the input bytes, in any order, are 0xF6, and 0x0F, the value in the register becomes 0x06). The orange cells are read-only, and contain device and firmware versions and some other parameters.

Some general setup can be performed on the operation of the device. The following can be configured:

- The I^2C address
- The ChipMode (Watchdog timer, TTL and Selector setup)
- Monotonic counter settings
- Key usage limits

[1] The Microchip DM320109: https://www.microchip.com/developmenttools/ProductDetails/DM320109
[2] Webinar on YouTube: https://www.youtube.com/watch?v=SFpkqmwoEeg



Configuration Zone

Configuration Zone - This zone has been read from the Device

	00	01	02	03
00	SN[0:1]		SN[2:3]	
04	RevNum			
08	SN[4:7]			
0C	SN[8]	Reserved13	I2CEnable	Reserved15
10	I2CAddress	Reserved17	OTPmode	ChipMode
14	SlotConfig00		SlotConfig01	
18	SlotConfig02		SlotConfig03	
1C	SlotConfig04		SlotConfig05	
20	SlotConfig06		SlotConfig07	
24	SlotConfig08		SlotConfig09	
28	SlotConfig0A		SlotConfig0B	
2C	SlotConfig0C		SlotConfig0D	
30	SlotConfig0E		SlotConfig0F	
34	Counter0			
38				

3C	Counter1			
40				
44	LastKeyUse			
48				
4C				
50				
54	UserExtra	Selector	LockValue	LockConfig
58	SlotLocked		Rfu90	
5C	X509Format00	X509Format01	X509Format02	X509Format03
60	KeyConfig00		KeyConfig01	
64	KeyConfig02		KeyConfig03	
68	KeyConfig04		KeyConfig05	
6C	KeyConfig06		KeyConfig07	
70	KeyConfig08		KeyConfig09	
74	KeyConfig0A		KeyConfig0B	
78	KeyConfig0C		KeyConfig0D	
7C	KeyConfig0E		KeyConfig0F	

Figure 4: The Configuration Zone of an ATECC508a or ATECC608a

- Lock states - one-time-changeable!

One Time Programmable Zone

A further one-byte configuration slot, the OTPmode, configures the One Time Programmable Zone use scenario, with two choices: read-only (0xAA) and consumption mode (0x55). The latter configures the entire register to a writable register, where the register is altered through a bit-wise AND comparator, as previously described for the lock states. All 64 bytes in the OTP Zone start at 0xFF (all bits are set to 1), unless otherwise configured. The changes to the register itself take effect after enabling the data and OTP lock.

Data Zone

Lastly, the slot and key configuration can be changed, with two bytes for the slot configuration and properties, and two bytes for the encryption key details, like the type of encryption. ACES allows interactive configuring of these bytes:

SlotConfig

The Slot Configuration bytes are the second block of 32 bytes in the Configuration Zone. These, in sets of two bytes, dictate the behaviour of key slots after locking. The settings include:

- Write permissions
- Data encryption key pointers
- Secrecy settings
- Read permissions
- Message Authentication Code permission

This also means that if configured incorrectly, the key is not a secret, and can be fetched by a set of commands. In the similar fashion it is possible to overwrite slots.

KeyConfig

The Key Configuration bytes are the fourth or last block of 32 bytes in the Configuration Zone. Similar to the Slot Configuration, these apply to a single slot only, and are locked in after the configuration lock. The settings include:

Figure 5: The interactive configuration window provides a clear overview of the settings and provides the accompanying configuration bytes in hexadecimal format.

- Private/Public key selection and formatting
- Intrusion latch response
- Individually lockable or group lock
- Public key generation permission
- Authorization requirement setting

Configuring the ExpLoRer's ATECC chip

Note: To prevent the accidental locking of zones and slots, this is written in separate functions (locking the individual slots is not encouraged for the tutorial: only in an application would this be required).

Now that we have a basis understanding of the chip itself, it is possible to start with some basic functionality. Using the Microchip CryptoAuthLib library, most commands start with:

```
atcab_
```

These commands can be found in the 'basic' folder in the CryptoAuthLib library. These functions take a lot of intermediate steps out of the process to allow for a lower learning curve. Furthermore, there are a few commands that can be used without engaging the lock state. Some of these are used in further examples:

```
atcab_wakeup()                  // Wake-up from sleep
atcab_sleep()                   // Sleep command
atcab_init(gCfg)                // initialize device using gCfg configuration
atcab_read_serial_number(sn)    // stores serial number into sn array
atcab_info(rev_info)            // stores revision into rev_info array
atcab_is_locked(zone, &isLocked) // store boolean lock state of zone in isLocked
```

```
atcab_write_config_zone(CONFIG)      // writes CONFIG to config zone
atcab_read_config_zone(read_buf)     // stores config zone into read_buf
atcab_random(rand_num)               // stores a 32-byte random number in rand_num
```

With some of these basics in mind, the configuration can be written. Again, with the Microchip ACES Crypto Studio, this is more interactive for achieving the required configuration. The configuration is compiled. The following settings are preset for this tutorial:

```
// For the ATECC608a:
0x01, 0x23, 0x00, 0x00, 0x00, 0x60, 0x00, 0x04, 0x05, 0x06, 0x07, 0xEE, 0x01, 0x01, 0x00,
0xC0, 0x00, 0xA1, 0x00, 0xAF, 0x2F, 0xC4, 0x44, 0x87, 0x20, 0xC4, 0xF4, 0x8F, 0x0F, 0x0F, 0x0F,
0x9F, 0x8F, 0x83, 0x64, 0xC4, 0x44, 0xC4, 0x64, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F, 0x0F,
0x0F, 0x0F, 0x0F, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0xFF, 0x84, 0x03, 0xBC, 0x09, 0x69, 0x76, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x0E, 0x40, 0x00, 0x00, 0x00, 0x00,
0x33, 0x00, 0x1C, 0x00, 0x13, 0x00, 0x1C, 0x00, 0x3C, 0x00, 0x3E, 0x00, 0x1C, 0x00, 0x33, 0x00,
0x1C, 0x00, 0x1C, 0x00, 0x38, 0x10, 0x30, 0x00, 0x3C, 0x00, 0x3C, 0x00, 0x32, 0x00, 0x30, 0x00
```

The ATECC508a has a very similar configuration data set, but differs due to the update in features for the ATECC608a (the equivalent data set is also programmed in the configuration program).

The first 16 bytes are permanent device-specific data, and can be ignored for the purpose of programming the ATECC. Byte 16 (index starts at 0), or the first byte on the second line, is the I^2C address that is to be used by the device; only useful if another device is attached with the same address.

Example Slot Configuration

Further along the second line are the first slot configuration bytes (bytes 20 and 21); the first being '0xAF, 0x2F'. These are the slot configuration bytes for slot 00, and tells us that the slot is configured to prevent writing when the slot is locked, and is a secret (no reading is ever permitted!). When the slot is locked, is becomes single-use: one operation is permitted with the key inside.

The key configuration that is applied to this slot is '0x33, 0x00', which tells us that the slot is an ECC P256 type key. It is private (not public), lockable, and can be used to generate the associated public key. Further is defined that this key is permitted to be used for the signing of data, both externally and internally provided. Lastly, the key is permitted to perform a Diffie-Hellman key exchange (requires a public of the other end), and stores the premaster secret (or shared secret) generated from this key pair into slot N+1 (or slot 01). It is also possible to have this premaster secret returned as a response to the ECDH command.

Configuration Example

The provided configuration sketch will configure the ATECC chip. The configuration will not proceed if it does not detect a serial monitor connection. The serial prints various debug messages and communication results, including the device specific lines such as revision number, detailing the ATTEC model, and the lock state.

Before configuring, most commands either return an error state, or a test output to identify correct functioning: the random number generator returns a predefined set of high (0xFF) and low (0x00) bytes.

This configuration is identical to the configuration used in Microchip's "Test" folder in the library, and is capable of supporting (almost all) functionality of the ATECC chips.

Using the ATECC

The examples provided are very self-explanatory, this section merely supports reading and running the programs!

Once the chip is configured and the configuration zone is locked, it is possible to access the previously unavailable functions: such as the (true) random number generator. This generator outputs a 32-byte true random number, and outputs this as response to the command:

```
atcab_random(rand_num)           // Stores a 32-byte random number in rand_num
```

This is shown in the TRNG example code provided.

Similar to before the lock was applied, the entire configuration can be read out (with the previously stated *atcab_read_config_zone(read_buf)* command), different is that any of the key slots for which read operations are permitted can be read as well. Write operations can, under similar circumstances, be performed as well.

Additionally, a few sets of actions can now be performed:

- Elliptic Curve Diffie-Hellman key exchange using a private-public key pair (ECDH).
- AES 128-bit symmetric encryption (the ATECC508a is not capable of this).
- Encrypted Read and Write operations.

Encrypted Read/Write Example

Encrypted reading and writing is an interesting process: it encrypts the data communicated between the master MCU and the ATECC chip, where it is then stored as plain text in a slot. This slot must be setup for encrypted read and write operations. In the provided configuration example, key slot 4 is enabled as a key that can be written to the slot externally (as required for a symmetric encryption), but not read.

The example provided goes through chip initialization, which is a software setup on the MCU. Then, for the configuration provided in the configuration example, the key used for the encryption and decryption of data is named ”SLOT_4_KEY” and is written into slot 4, as the name implies. The data slot used for the encrypted read-write example, is slot 8. This slot is configured to only accept on-device keys for the encryption, and the key slot must be configured for this operation.

Warning: if slot 4 is not locked, a hacker may input their own key and use this to read the contents: locking is thus necessary if this is used in an application. Similarly, if another slot can be used for the encrypted read/write functions, it too must be locked, as that slot can then be used instead of slot 4.

The example also provides a ’fake’ key, with only the first byte altered. As expected, the encryption on-chip is performed using the previously implemented key, but cannot be correctly decrypted using this fake key.

The most notable commands from the library, that are used are:

```
// to write the encryption key to slot 4, and the data to write in data
atcab_write_zone(zone, slot, block, offset, data, length)
// to write the data encrypted to slot 8, and data to write in data
atcab_write_enc(slot_w_data, block, data, enc_key, slot_w_key)
// to read the data decrypted from slot 8, and 'data' to store the returned data in
atcab_read_enc(slot_w_data, block, data, enc_key, slot_w_key)
```

ECDH Example

In the provided ECDH example, two ’identities’ are set up: Felix, and Tom. These identities wish to communicate using a shared secret, but do not wish to communicate the shared secret for the fear of eavesdropping. They both then choose their own (private) ECC 256-bit key, and using this, derive the public 512-bit dependent key, and share the latter with each other.

Combined with the secure element feature, the 256-bit private key cannot be practically obtained, ensuring security for the future transactions between Felix and Tom.

In order to show the functionality of ECDH, however, both private keys must be stored on the same chip, but both are in fact never known to the owner of the device. The example uses, as always recommended for this kind of application, the true random number generator to generate the secret private keys for both. These are stored in slots 0 and 2, and the public keys are retrieved on the serial monitor.

The commands used are:

```
atcab_genkey(priv_slot_id, pubkey) // stores the 64-byte public key into pubkey
atcab_ecdh(priv_slot_Felix, pubkey_Tom, ECDH_Felix) // reads pubkey_Tom for the public key
                          // and stores the ECDH premaster (shared) secret in ECDH_Felix
atcab_ecdh_enc(priv, pub, ecdh_key, slot_key, slot_w_key) // same, but combines encrypted read
                                                          // of slot priv
```

AES Example

This last example shows how to use AES 128-bit encryption with the ATECC608a. This symmetric encryption method uses a 16-byte key to encrypt a 16-byte message.

The example provides a set of 4 AES-128 keys, in addition to 4 plain-text messages that are to be encrypted. These 4 messages each have a set of data detailing what the returned encrypted message should look like, for each of the 4 provided AES keys (16 encrypted forms in total), in order to provide testing of the functionality.

There are a few modes of AES encryption. The one showed is AES ECB (or Electronic CodeBook), the simplest of the set of forms, and easiest to crack (but still very secure - gaining access to the encryption key is the fastest method).

Other encryption modes that the ATECC608a is capable of are: CBC, CTR and GCM and alter the encryption based upon previous outputs; these are not recommended in LoRa applications since the successful implementation is entirely dependent on the message arriving, and if so, intact: losing one part of a message or a message as a whole results in the loss of the capability to decrypt the transmitted messages after the loss. It is possible to use these if the method is reinitialized for every message transmitted.

Note: the nonce commands are used to put the symmetric encryption key in the TempKey buffer, for use in the encryption process. This is not a safe storage space as it is overwritten by some commands and freely accessible.

The commands used are:

```
atcab_nonce(num_in)                    // 32-byte num_in to put in the TempKey register
atcab_nonce_rand(num_in, rand_out)     // 32-byte rand_out is result of TRNG used with nonce
atcab_nonce_load(target_buffer, num_in, num_size)

// AES commands require 16-byte key, 16-byte input message, 16-byte output register
atcab_aes_encrypt(key_slot, block, plain_in, cipher_out)
atcab_aes_decrypt(key_slot, block, cipher_in, plain_out)
```

Beyond the Examples

The examples are a fixed set showcasing some of the capabilities of the crypto chip, and a thorough understanding of the chip's commands is required to move beyond the basics. To name most of the possibilities that are not provided in the examples:

- AES CBC: cipher Block Chaining (key dependent on previous message)
- AES CTR: Counter (another key obscuring method)
- AES CMAC: Block cipher Message Authentication Code method
- Message signing and verifying
- Key derivation
- SHA hashing
- HMAC: Hashing Message Authentication Code method
- Secure Boot: firmware and software source authentication
- Message digests
- Limited-use keys, using one of two monotonic incremental counters