WSC332: Cybersecurity for Embedded Systems Report

F130812

1. Introduction

The need for cybersecurity has become critical in embedded applications with implementations in autonomous vehicles and drones, consumer electronics and vital control systems – especially with the rising number of cyberattacks. The problem also comes with the need for cybersecurity on resource-constrained devices, requiring unique and bespoke solutions for robust security.

This report investigates the use of lightweight cryptography algorithms for the STM32767ZI microcontroller, with focus on performance on deployment of these algorithms, leading to a case study for a simulated drone environment. The project will explore performance of encryption algorithms, the use of SHA-256, RSA verification and a combined solution featuring all.

2. Algorithm Overview

This project uses XTEA and Blowfish symmetric algorithms, SHA256 hash functions and RSA digital signatures. Each example is then tested for memory footprint and execution time using different deployment strategies.

2.1 Blowfish

Blowfish is a symmetric encryption algorithm that works on 64-bit blocks of data with key lengths from 32 to 448 bits. In this coursework, a 128-bit key is used with Blowfish and it is applied to encrypt things like the drone's unique ID, a randomly generated vehicle licence plate and simulated GPS position data in the Beakman challenge, further detailed in Section 7.

2.2 XTEA

XTEA (eXtended Tiny Encryption Algorithm) is a lightweight encryption algorithm optimised for speed and simplicity, as will be seen in this report. It also works using 64-bit blocks and uses a 128-bit key. Although, the XTEA algorithm has a much simpler encryption process than Blowfish.

XTEA was used in Part 1 to compare with Blowfish. Its applicability for embedded systems is showcased with how little memory it takes up. Though for better security, other algorithms would be preferred.

2.3 SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function which maps input data to a 32-byte digest. It is used for integrity generation to ensure that no modification to the firmware has been made. This is verified by recalculating and comparing the hash.

Throughout this coursework SHA-256 was applied to verify firmware integrity. The STM32767ZI board executes the `FW_Hash_Verify()` from the X-CUBE-CRYPTOLIB library. This command is performed at boot and computes the hash of the firmware and compares it to a reference digest stored in flash memory.

To ensure that the hash checks match, the Host_Tools utility was used, specifically `PostBuild_hash.bat`.

This new binary file was then flashed to the STM32767ZI board using STM32CubeProgrammer to program the board with the correct hash value.

This technique mirrors the secure boot functionality found in real embedded systems to protect against malicious firmware modifications without the need for external hardware.

2.4 RSA and ECDSA

To explore asymmetric cryptography algorithms, RSA and ECDSA were considered. These use two separate keys; a public key to be shared among many, and a private key which must be kept secret. These asymmetric algorithms were considered:

- RSA is based on the difficulty of factoring large numbers and is therefore widely used in traditional systems
- ECDSA uses elliptic curves to security with smaller keys and better efficiency and is more suitable for resource-constrained platforms.

Due to lower computational and memory requirements, ECDSA was chosen for this project. Details of the attempted integration is described in Section 4.

3. XTEA, Blowfish and SHA256 Analysis

To evaluate the performance of these symmetric encryption algorithms, as well as the algorithms with firmware integrity checks, four configurations were benchmarked:

- XTEA
- Blowfish
- XTEA + SHA256

- Blowfish + SHA256

Each configuration was compiled using -O0 (no optimisation) and -O3 (full optimisation) flags, with execution time measured over 100 iterations using the TIM11 hardware timer. Memory utilisation was benchmarked using the STM32CubeIDE build outputs (Flash + RAM).

3.1 Comparative Results Table

Model	Compilation Flag	Avg. Execution Time (µs)	Memory Utilisation	
			Flash (KB)	RAM (KB)
XTEA	O0	77.46	28.82	2.58
	O3	27.52	23.63K	2.58
Blowfish	O0	30.72	33.66	6.69
	O3	5.42	32.01	6.69
XTEA + SHA256	O0	147.50	34.83	2.75
	O3	96.11	29.66	2.75
Blowfish + SHA256	O0	176.76	39.49	2.75
	O3	89.78	37.83	2.75

Table 1 - Metrics for XTEA and Blowfish Using Different Compilation Flags

3.2 Performance Insights

3.2.1 XTEA vs Blowfish:

XTEA was consistently outperformed by Blowfish across compilation flags and implementation of SHA256, respectively. It would be expected that XTEA would outperform Blowfish, but the implementation strategy taken could have offset this.

3.2.2 Impact of SHA256 integration

Adding firmware integrity through SHA256 significantly increased the runtime for both algorithms. Though this is to be expected with the added complexity of SHA256 added to each program.

3.2.3 Effects of Optimisation

Compiler optimisation consistently showed to improve the execution time across all four configurations, with Blowfish showing a 5.6x improvement in performance, while SHA256 applications showed more modest improvements of ~1.7x.

3.2.4 Memory Footprint

XTEA applications offered the least memory usage across all configurations, showing that it would be the most feasible for memory-constrained devices. Blowfish added some overhead flash and more RAM with SH256 having a more significant increase in memory usage needed.

3.2.5 Final Insights

XTEA has shown that it is the best cryptographic algorithm for memory-constrained devices, whilst also offering decent speed. Blowfish is a better option if memory is not a concern as it offers itself as a stronger algorithm with faster execution time.

SHA256 hashing shows to impact runtime and must be selectively applied, for instance only at boot of a device or not at all depending on the use case.

Regardless, the microcontroller used here very easily handled these tasks without great strain, demonstrating the clear feasibility for cryptography algorithms in embedded cybersecurity applications.

4. ECDSA Digital Signature Verification (Attempt)

To add another level of security to the firmware, an attempt at implementing ECDSA (Elliptic Curve Digital Signature Algorithm) was made. This would mean that the software would be signed by a trusted authority to ensure the software is authenticated. This signature ensures that unauthorised or malicious updates to the firmware would not take place.

To implement this, a procedure of using OpenSSL to generate a P-256 keypair was used. The private key stored as `private_key.pem`, generated from the firmware binary, is to be used to sign the SHA256 digest and the corresponding `public_key.pem` would be embedded in the device's firmware to verify the authenticity of the firmware.

After then compiling the firmware binary, ECDSA signing operations were applied to it using OpenSSL. This produced a "`firmware.sig`" to be added to the existing firmware image.

On the STM32767ZI platform, signature verification was to be applied using STM32's X-CUBE-CRYPTOLIB library using an API for ECDSA verification. The intended process was to first compute the firmware hash using `FW_Hash_Verify()`, then read the stored signature from flash memory and call the verification routine from the public key.

Despite succeeding in generating public and private keys and obtaining the firmware signature, some issues arose during the integration of them to the board. The main obstacle was getting the cryptographic middleware to work on the STM32F7 platform. The cryptolib packages produced various incompatibility and missing elements errors.

Lack of time and clear documentation on deploying ECDSA to the STM32F7 platform using STM32's X-CUBE-CRYPTOLIB meant that the full integration of this process was failed to be completed. Thus, the final implementation and the one carried forward is limited to SHA256 firmware integrity verification only.

5. Drone Casestudy

The next exercise involved the implementation of various cybersecurity techniques for the STM32F767ZI platform to demonstrate how they can be used for a resource-constrained, security-critical scenario. The mission is to emulate an autonomous drone mission that transmits encrypted data, an operation that could be found in real-world drone deployments.

5.1 Hash Embedding via Host Tools

To begin the security measures, the drone initiates with a SHA256 hash check to ensure the authenticity of the running firmware as done in the initial part of the coursework. This is performed using the `FW_Hash_Verify()` routine. If the hashes were to not match, execution of the program is halted, emulating a secure boot mechanism.

This parallels real systems where integrity validation can be found in secure drones and IoT devices, to ensure that the firmware has not suffered malicious tampering which could compromise the systems. This software implementation mimics the hardware-accelerated bootloaders seen in many real devices.

5.2 Drone Authentication Using Encrypted UID

After verification of the firmware, the system proceeds to identify the drone through its unique ID. This simulation was done using the 96-bit embedded identifier embedded in the STM32 board at factory level. This ID is accessed using inbuilt HAL functions and is then split across two 64-bit Blowfish blocks for encryption. The encrypted ID simulates a scenario where a real-world drone would need to authenticate itself to a base station without exposing its identity to potential attackers.

```
uid[0] = HAL_GetUIDw0();

uid[1] = HAL_GetUIDw1();

uid[2] = HAL_GetUIDw2();

...

Blowfish_Encrypt(&bf_ctx, &uid_block1[0], &uid_block1[1]);

Blowfish_Encrypt(&bf_ctx, &uid_block2[0], &uid_block2[1]);
```

5.3 Encrypted Vehicle Plate Transmission

The firmware of the drone next simulates the scanning of 6-digit numberplates collected through surveillance missions. These plate numbers are generated through the STM32's random number generation hardware peripheral. This number plate is then encrypted using Blowfish.

The encryption of mission elements ensures secrecy in scenarios where drones may be communicating through untrusted/public channels, for instance over radio channels or Bluetooth.

5.4 Secret Message Transmission

Lastly, the drone is required as part of its mission to encrypt and send message data to the ground station. A 16-byte is encrypted using Blowfish and decrypted later to reveal a message reading, “Amazing Ost \o/”. This encrypted and decrypted data is logged, verified for its correctness and outputted mimicking exchange of private information such as mission plans, control instructions or position.

6. UART-Based Encrypted Data Transmission and PC Decryption

To ensure a secure communication of the encrypted data between the drone (STM32767ZI board) and the host ground station (PC), UART was used to transport the encrypted data from the drone. This means that data is kept fully secure until it reaches the groundstation and reduces the processing required by the drone. The data was received and then decrypted using a Python-based script.

6.1 Encryption and UART Transmission

Onboard the drone, Blowfish encryption is performed on these sensitive messages. Encrypted 64-bit blocks are then transmitted over the UART. The UART is configured 115200 baud and the HAL_UART_Transmit is used to ensure all data is securely sent.

6.2 Python Ground Station Receiver

A Python script running on the Ground Station opens a serial connection through pyserial, and incoming blocks of ciphertext are received. Upon receiving the ciphertext, each block is parsed into 32-bit blocks and are processed by the exact same Blowfish algorithm used on the STM32 board.

To ensure compatibility with the blowfish.c and blowfish.h files used on the board, these files were compiled into a Windows DLL (blowfish.dll) using a C wrapper. This was done using MSYS2 shell through this command:

```
gcc -shared -o blowfish.dll -DBUILD_DLL -fPIC blowfish.c
```

This allowed for the Python script to load and execute the exact same Blowfish instructions, thus guaranteeing identical behaviour and successful decryption.

6.3 Ground Station Decryption

The Ground Station setup was tested by verifying the decryption of the encrypted UID blocks, the encrypted randomly generated number plates and the secret mission message. All decrypted results from the Python script exactly matched the original values generated by the drone.

The UART transmission of encrypted values and external decryption simulates a realistic drone-to-ground station security route in which microcontrollers encrypt data and base stations validate and decrypt data.

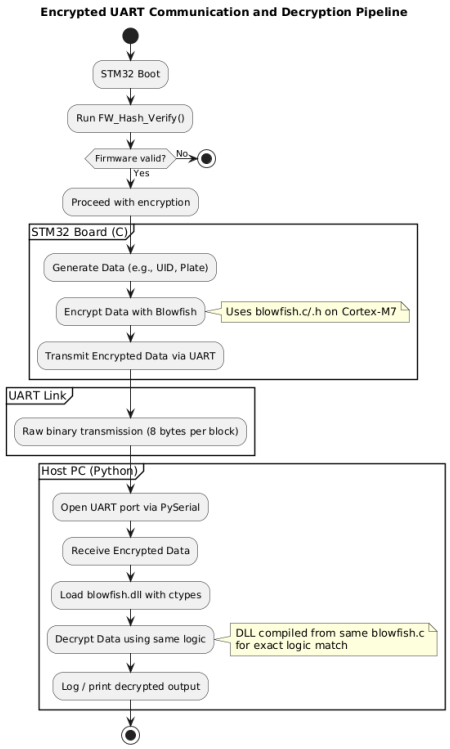

Figure 2 - Encrypted Pipeline for Encryption of Data

```
Received: >>> FW Hash OK
Received: >> Drone ID phase (HAL UID Read)
Received: UID (96-bit): 38313533 34315105 002d0037
Received: Encrypted UID block 1: {9e351c54 6d7089ec}
Received: Encrypted UID block 2: {d39116d0 5660b897}
[RECONSTRUCTED UID] 38313533 34315105 002D0037
Received: >> Car-plate phase
Received: Plain plate: 662155
Received: Encrypted plate: {2183097c 4f78360f}
[DECRYPTED PLATE] 662155
Received: >> Secret message phase
Received: Encrypted secret:
Received: block1: {69d17faa e9871f96}
Received: block2: {0aa48825 a33574f6}
[DECRYPTED SECRET] "Amazing Ost \o/ "
```

Figure 1 - UART Output of Ground Station Python Script

7. Beakman Challenge

To demonstrate the ability of cryptographic techniques for real-time mission conditions, the Beakman Challenge was established wherein an extended encrypted telemetry was continuously transmitted and decrypted live on the Ground Station.

7.1 Real-Time Telemetry Loop

The drone captures (randomly generates) sensor data during flight. At a one-second frequency, the system:

- Updates roll, pitch and yaw
- Adjusts latitude/longitude and altitude
- Simulates battery drain

- Encrypts all of these messages using Blowfish
- Transmits encrypted blocks via UART

```
uint32_t pos_block[2] = { lat, lon };

Blowfish_Encrypt(&bf_ctx, &pos_block[0], &pos_block[1]);

printf("TELE_POS: {%08lx %08lx}\r\n", pos_block[1], pos_block[0]);
```

Similar encryption was performed for Tele_ALT, TELE_RP, TELE_YAW and plate.

7.2 Python Receiver

On the Ground Station a Python Script much like the one Part 6 is used for receiving data through the UART and decrypting the data through the same Blowfish algorithm used for encryption.

```
decrypted = decrypt_block(high, low)

alt = int.from_bytes(decrypted[:4], 'little')

batt = int.from_bytes(decrypted[4:], 'little')
```

7.3 Mission Results

The telemetry stream is decrypted and visualised in real-time, with consistent output frames reflecting the changing drone status:

TELEMETRY FRAME

Position : LAT 5123600 LON 1342250

Altitude : 43 m

Roll : 0.99°

Pitch : 4.0°

Yaw : -32.0°

Battery : 13 %

Scanned number plates were also logged every cycle:

SCANNED PLATE

Plate : 114244

Upon reaching 10% battery level, a descent sequence is triggered returning the drone back home.

7.4 Real-Time Visualisation of Encrypted Telemetry

The data decrypted by the Ground Station was plotted in real-time using Python and Matplotlib, providing a visual view of the drone's simulated behaviour. Altitude (m), Battery (%) and Yaw (°) were plotted over time.

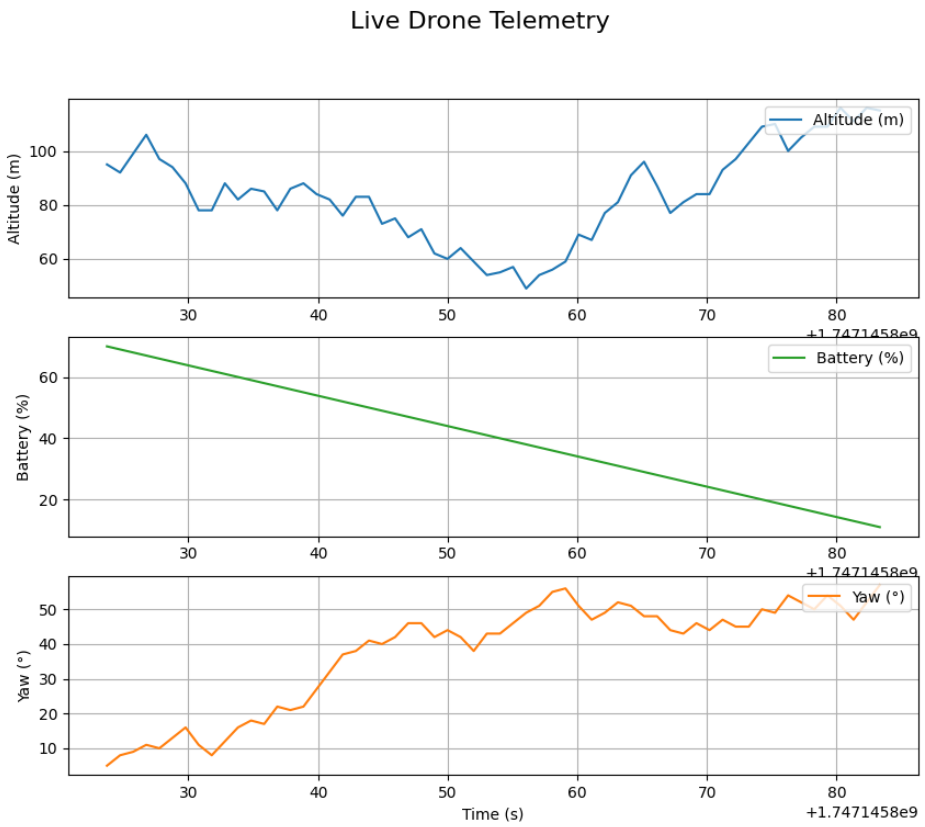

Figure 3 - Live Decrypted drone telemetry plotted from UART input using Blowfish .dll

8. Conclusion

This report explored the deployment and performance of Blowfish and XTEA, two lightweight symmetric cryptographic algorithms on the STM32767ZI. Hash functions for boot security and an attempted integration of asymmetric signature verification (ECDSA) was done to demonstrate the feasibility of implementing real-time embedded

security on resource constrained platforms. To showcase how these techniques can be used in real-world applications, a drone and ground station scenario where encrypted data was transmitted through UART was developed.

The performance benchmarking showed that Blowfish consistently outperformed XTEA in execution time although had slightly worse memory trade-offs. Compiler optimisations showed to have significant impact, particularly for Blowfish with a 5.6× speedup measured. The SHA-256 hashing technique was added to validate the firmware integrity at boot using the `FW_Hash_Verify()` function to show a secure boot process.

Unfortunately, the integration of the full ECDSA signature could not be achieved due to middleware problems but it highlighted the importance of authenticated firmware and showed a promising route for future developments.

The drone case study tied these different cryptographic techniques together to show a practical application with the addition of secure transmission of sensitive information and data. The system simulated a realistic drone-to-ground station communication pipeline through transmitting encrypted data from the board over UART and decrypting it externally through a python script using a shared Blowfish DLL. The Beakman challenge highlighted the robustness and scalability by executing repeated telemetry cycles and including a visualisation of transmitted metrics.

In conclusion, the STM32F767ZI platform showed that it is fully capable of supporting end-to-end cryptographic operations for a simulated drone environment. This project not only validates the feasibility of the use of lightweight cryptographic algorithms for STM32 platforms but also shows a complete pipeline of secure data generation, encryption, decryption, transmission and verification – a very supportive design for real-world embedded cybersecurity applications.