http://dx.doi.org/10.14236/ewic/icscsr19.14

RESTful HTTPS over Zigbee: Why and how?

Soo Yee Lim
Bristol Cyber Security Group
University of Bristol
Bristol, UK
sl16004@bristol.ac.uk

Joseph Gardiner
Bristol Cyber Security Group
University of Bristol
Bristol, UK
joe.gardiner@bristol.ac.uk

Barnaby Craggs
Bristol Cyber Security Group
University of Bristol
Bristol, UK
barney.craggs@bristol.ac.uk

Awais Rashid
Bristol Cyber Security Group
University of Bristol
Bristol, UK
awais.rashid@bristol.ac.uk

With the advent of the Internet-of-Things (IoT), there has been a wave of wireless protocols aimed at providing communication between connected devices. One of the most widely used protocols is Zigbee, a derivative of the IEEE 802.15.4 protocol for building low power mesh networks. Whilst Zigbee has largely found a use in consumer grade devices, it has also been explored as a protocol for use in industrial and building automation systems. A number of vulnerabilities have been found within the Zigbee and IEEE 801.15.4 standard which could damage the integrity of the transmitted data. Therefore, we propose a solution wherein a RESTful HTTPS protocol is transmitted over Zigbee networks, effectively providing a double layer of security. We demonstrate that RESTful HTTPS over Zigbee is possible, and evaluate its performance.

1. INTRODUCTION

Zigbee is a popular protocol used for multiple applications, in particular for Internet-of-Things (IoT) devices. Zigbee, defined by IEEE 802.15.4 (13), is designed to be low cost, low data rate and low power, with networks formed by a mesh of wireless devices. Many commercial home IoT products make use of Zigbee for communication; for example, it is the primary communication method used within the Phillips Hue lighting system.

Whilst Zigbee is currently not widely used in industrial settings, this is set to change. The Zigbee Alliance themselves identify industrial control and building control as potential Zigbee applications (10). As stated by Egan (10), for industrial and building management systems, the two primary concerns for industrial and building automation systems is robustness and security. Robustness is provided by its wireless mesh structure; a Zigbee network can rebuild routes when nodes are removed from the network, providing robustness. The use of 128 bit AES encryption, with keys distributed through a trusted medium on install, provides some guarantee of security. This is backed up by Rault and Malik (20), who provide an analysis of the performance of Zigbee for industrial networks. Castro et al. (5) propose a process automation system for refrigeration which operates over Zigbee. Whilst other wireless protocols such as WirelessHART (6) have seen widespread adoption in industrial settings, these are substantially more expensive than Zigbee compatible devices, meaning that many may look to Zigbee for more cost-effective solutions.

However, there are a number of vulnerabilities in the Zigbee protocol itself, and underlying IEEE 802.15.4 standard (14; 1; 29; 26). Therefore, we argue that Zigbee on its own is not suitable for scenarios where sensitive or safety critical data might be transmitted over the Zigbee network.

To combat this issue, we propose the use of a RESTful HTTPS protocol over Zigbee networks. Through the use of a HTTPS interface, as well as the encryption provided by Zigbee, traffic over the wireless network is in effect double encrypted. This means that, if the security of the Zigbee network is compromised either through a known or to be found vulnerability, the traffic maintains the guarantees provided by the TLS connection underneath. We measure the performance of this double encryption based on increasing payload sizes, and find that, for smaller payloads, RESTful HTTPs over Zigbee is feasible.

Our main contributions are as follow:

- We argue the need for RESTful HTTPS interfaces over Zigbee
- We provide a proof of concept implementation.
- We evaluate the performance of RESTful HTTPS over Zigbee.

The rest of this paper is structured as follows: In Section 2 we provide a background on RESTful interfaces and Zigbee networks. In Section 3, we identify vulnerabilities in the Zigbee standard from the literature. We then provide an overview of how we implemented a RESTful HTTPS interface over Zigbee in Section 4. Section 5 provides an evaluation of the performance of RESTful HTTPS over Zigbee, and finally we conclude and indicate future expansion in Section 6.

2. BACKGROUND: RESTFUL INTERFACES AND ZIGBEE

2.1. Representational State Transfer (REST)

Representational State Transfer (REST) is a network-based architectural style developed in parallel with HTTP 1.1. REST defines a set of constraints that attempt to minimize latency and network communication while maximizing the independence and scalability of component implementations at the same time (12). For this reason, RESTful architectures have become increasingly popular in distributed systems.

The six architectural constraints are as follows (12):

1. Client-Server — A server component is the provider of services or resources; a client component is the service requester. The principle behind this constraint is the separation of concerns. Separating functionalities simplifies server components for better scalability and improves portability of client components across multiple platforms. This separation also allows each component to evolve independently.
2. Stateless — This constrains all communications to be stateless in nature. As a result, visibility is improved because the full nature of the request can be determined with a single request datum. Scalability is improved because server components can quickly free resources, which further simplifies implementation. Reliability is improved because this constraint eases the task of recovering from partial failures (27).
3. Cache — The data within every response must be labelled as cacheable or non-cacheable. The advantage of this is that some client-server interactions can potentially be partially or completely eliminated, further improving efficiency and performance.
4. Uniform Interface — This is the key constraint in the design of any RESTful system. The four interface constraints are:
   (a) Identification of resources — REST resource is an abstraction of information. Resources have to be identified by resource identifiers in requests.
   (b) Manipulation of resources through representations — Client components holding a representation of a resource have enough information to modify or delete the resource on the server component.
   (c) Self-descriptive messages — Each message has enough information to describe how it should be processed.
   (d) Hypermedia as the engine of application state — Client components should be able to discover other resources using the server-provided links included in the response.
5. Layered System — The system architecture needs to be composed of hierarchical layers, with the inner layers only visible to adjacent outer layers.
6. Code-On-Demand [optional] — The functionality of a client component can be extended by downloading executable code in the form of applets or scripts.

2.1.1. RESTful HTTP

Roy Fielding (11) has stressed that a HTTP-based interface does not necessarily imply that it is RESTful. REST improves system scalability, and without it, horizontal growth would be practically difficult. Scalability is critical in ICS because support for a significant number of components and interactions amongst those components in the operational environment may be required. As such, it is a good practice to make sure that HTTP interfaces strictly adhere to all REST constraints listed in Section 2.1.

2.2. Zigbee

Zigbee is a low-cost, low-rate, low-power, wireless mesh network standard targeted at automation and control applications (7). The physical layer and MAC layer of Zigbee are defined by the IEEE 802.15.4 standard. The Zigbee Alliance then builds on this

basis by providing the security service provider, network layer and the framework for the application layer. The application framework is the environment where up to 254 application objects, i.e. endpoints, are simultaneously hosted on the Zigbee device. The IEEE 802.15.4 standard is designed for industrial applications with very low power consumption and relaxed needs for data rates (13). By building on IEEE 802.15.4, Zigbee naturally inherits the desired low rate and low power consumption features. As Zigbee's low power consumption has limited its transmission distances to 10 - 100 meters, a multi-hop self-organised network topology is essential for transmitting data over long distances. Within Zigbee networks, devices are classified into three types:

Coordinator A full function device (FFD) that controls the entire network, relays messages and authenticates new nodes.

Router A full function device (FFD) that communicates with the coordinator and end device to relay and forward data packets.

End device A reduced function device (RFD) that can communicate with the router and coordinator, but cannot relay data from other end devices. This reduced functionality allows for the potential to reduce their cost because end devices do not have to stay awake all the time.

Zigbee networks will form one of three topologies. In the star topology, all end devices and routers are connected to a single coordinator node. The primary disadvantage of this topology is that the coordinator represents a single point of failure (SPoF). In addition, as all communication goes through the central coordinator, the throughput and range of the network are constrained by the coordinator. In the cluster tree topology, a coordinator forms the root node of a tree. Geographical expansion of the network is made possible by adding router nodes but this also means that if a router fails, portions of the network can become disconnected as there is no alternative route. Finally, the Mesh topology is one of the most flexible Zigbee structures. Unlike Cluster Tree topologies, a router can communicate with any other routers and end devices that are within its range. Hence, the network can recover from a node failure because there exist many different routes to a given node.

2.2.1. Security

Zigbee's level of security depends on the safekeeping of the symmetric keys, the protection mechanism, implementation of cryptographic mechanisms and relevant security policies. Therefore, its security architecture places emphasis upon the methods for

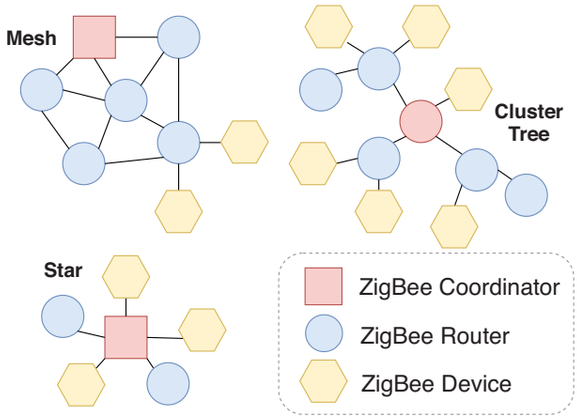

Figure 1: *Zigbee topologies*

key establishment, key transport, frame protection, and device management (2).

Due to cost constraints, Zigbee contains security assumptions that lead to an open trust model for a device. In other words, all applications running on a Zigbee device and different layers of the Zigbee stack trust each other. However, the major disadvantage of the open trust model is its vulnerability to insider attacks (19). As the security specification cannot be easily modified, this has motivated us to look into double encryption for its potential to solve this issue.

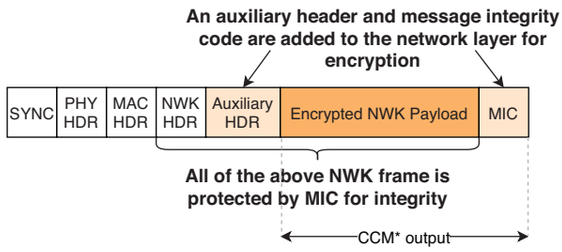

Figure 2: *A data frame with security on the NWK level*

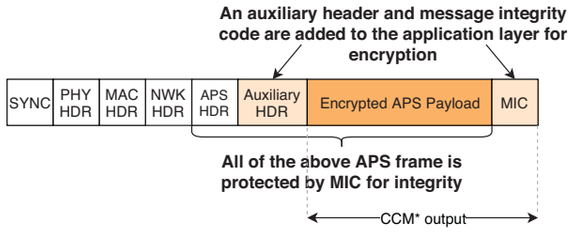

Figure 3: *A data frame with security on the APS level*

As a result of the open trust model, each layer in the protocol stack maintains its own security without correlation with other layers. Therefore, the auxiliary header and message integrity code in the network (NWK) layer (Figure 2) are independent

from those in the application support (APS) sub-layer (Figure 3). The MAC layer security is defined by the IEEE 802.15.4 standard, while Zigbee defines the security mechanisms for the NWK layer and APS layer. Both the NWK layer and the APS layer are responsible for the processing steps required to securely transmit outgoing frames and securely receive incoming frames. In addition to that, the APS layer is also responsible for securely establishing and managing cryptographic keys.

Applying security to a NWK or APS frame corresponds to an instantiation of the AES-CCM* mode of operation. Zigbee adopts a 128-bit Advanced Encryption Standard (AES) encryption and it provides data integrity and encryption through a generic combined encryption and authentication block cipher mode (CCM*), which is adapted from the Counter mode with Cipher Block Chaining Message Authenticity Check (CCM). A Message Integrity Code (MIC), as seen in Figures 2 and 3, of up to 128 bytes is used to ensure message integrity. Replay attacks are prevented using a 32-bit frame counter, which is reset to zero every time the network key is updated.

For the purpose of trust management, Zigbee introduced a role called “Trust Center” that is trusted by other devices within the network. The Trust Center is responsible for key distribution for the purpose of network and end-to-end application configuration management. In a secure network, all devices in the network shall recognise exactly one Trust Center, and that Trust Center should be unique to the network. Three key types are used: a master key for establishing a secure connection between devices and the trust centre, a network key common amongst all device (distributed by the trust centre), and a link key, derived from the master key, used for securing communication between two devices. When operating in standard security mode, as set by the Trust Center, keys are distributed to devices in an unencrypted format. Keys are only transmitted in an encrypted state in high security mode, secured using the master key.

3. RELATED WORK

We briefly cover here a number of security vulnerabilities found within Zigbee, and more generally 802.15.4 networks. For a more detailed analysis of Zigbee and IEEE 802.15.4, we refer the reader to (14; 26; 23).

3.1. Security of Zigbee networks

Killerbee (29) is an attack framework for Zigbee devices produced by Joshua Wright. The Killerbee framework utilizes a custom firmware for AVR RZ Raven USB Sticks, and through the interface software, a number of actions are available. These include enumeration, sniffing, OTA key sniffers and replay attacks. Replay attacks are possible as the IEEE 802.15.4 standard has no replay protection provided, whilst the Zigbee protocol only adds minimal replay protection (14; 1). Olawumi et al. (18) propose a three layered attack against Zigbee networks using the Killerbee framework, with enumeration, packet sniffing and replay attack stages.

Vidgren et al (26) propose two novel attacks against Zigbee networks. The first, the end-device sabotage attack, the attacker introduces extra routers and coordinators into the network, which will broadcast responses to end device requests. As end devices will continue polling coordinators whilst receiving responses, they will quickly run out of power, shutting the device down. The second is a key sniffing attack using Killerbee, possible when the network is configured in standard mode.

3.2. Security of IEEE 802.15.4

As early as 2004, Sastry and Wagner (22) identified a number of vulnerabilities in the IEEE 802.15.4 standard. They identify several different vulnerabilities, covering IV management, key management and integrity protection. Many of the vulnerabilities center around the use of a nonce which is tied to the encryption key also being used as a message counter to protect against replay attacks.

Sharma et al. (23) describe attacks against wireless sensor networks, and in particular for IEEE 802.15.4 describe jamming (25) and packet modification through symbol flipping. Sokullu et al. (24) provide a further jamming attack utilizing GTS time slots, which are incorporated into the protocol to allow for collision free transmission. By causing collisions within this component, a DoS attack is performed.

O’Flynn and Chen (16) discuss a side channel attack against the AES-CCM used within the IEEE 802.15.4 stack. Focusing on an Atmel ATMega128RFA1 system-on-a-chip, they are able to recover encryption keys through side channel power analysis in under 60 minutes without modifying the device. O’Flynn (17) also discusses multiple targeted jamming attacking against IEEE 802.15.4, in which a man in the middle attack is effectively performed allowing the attacker to read messages, and then decide whether or not to jam them.

Wilhelm et al. (28) demonstrate an overshadowing based message manipulation attack. In the overshadowing attack, a stronger signal than the target is introduced. In the case of a collision, the stronger

signal will be processed by the receiver rather than the original target signal. Through this attack, they are able to successfully replace the last 8 bytes of a message with a 69% success rate. —

4. IMPLEMENTING RESTFUL HTTPS OVER ZIGBEE

4.1. Hardware

4.1.1. Compute device

Originally, the intention was to use Arduino Mega 2560 as processing units for the clients and gateway. The benefit of these devices is the easy availability of hardware add-ons for different wireless protocols through shields, low cost and easy prototyping capability. However, the Arduino Mega 2560 has a limited 8 kilobytes of SRAM, which is insufficient for performing a TLS handshake. To overcome these constraints, we use Siemens Simatic IOT2020 devices as compute nodes. These devices are designed for experimentation around industrial IoT. The devices are Arduino compatible, supporting most Arduino compatible shields, but have the added benefit of running a full Yocto Linux kernel, as well as Ethernet and USB interfaces. As can be seen in Table 1, these devices have substantially more resources than Arduino devices, with a minimal cost overhead.

Name	Processor	Clock Speed	Flash	SRAM
Arduino Mega 2560	ATmega 2560	16 MHz	256 kB	8 kB
Siemens SIMATIC IOT2020	Intel Quark x1000	400 MHz	8 MB	256 kB

Table 1: *Technical specification comparison*

4.1.2. Ethernet

To measure performance over a physical wired connection, two Simatic IOT2020 devices are used. The ethernet connection is provided through the built-in ethernet ports on the IOT2020 devices, with routing between provided by a Netgear GS305 Gigabit Ethernet switch.

4.1.3. Zigbee

Wireless communication between two Siemens SIMATIC IOT2020 devices is set up over the Zigbee protocol using two Digi Xbee S1 IEEE 802.15.4 RF modules (9) with trace antenna. The Xbee modules do not contain an internal processor, only acting as a sender/receiver by communicating with the compute device over a serial UART connection. The client Xbee module is connected to the IOT2020

Figure 4: *Zigbee experimental setup with two IOT2020 devices and two Xbee modules*

using an Xbee USB converter, whilst the server Xbee module is connected to the second IOT2020 using an Arduino Mega 2560 functioning as a USB converter, as seen in Figure 4. Note that the Xbee module for both the client and server are identical devices. Xbee modules communicate with each

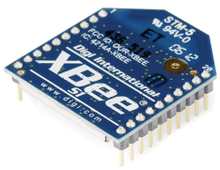

Figure 5: *Digi Xbee S1 802.15.4 RF module*

other wirelessly at a rate of 250 kbps through radio frequency (RF) at 2.4GHz. The Xbee modules have two operating modes: AT (Application Transparent) mode and API (Application Programming Interface) mode. In AT mode, the modules act as a serial line replacement, i.e. the data is packetized and sent exactly as it was received. Since we are not interested in just sending plaintext, the modules are configured to communicate in API mode, where data is communicated in API frames.

A PAN (Personal Area Network) is set up by configuring an Xbee device as a coordinator and the other as an end device. Since our coordinator is mains-powered, the network is set to non-beacon mode, meaning its receiver is on all the time, whilst the end devices are allowed to initiate a conversation at random intervals in non-beacon mode.The Xbee module supports 128-bit AES encryption. When encryption is enabled, the packet's payload is encrypted using the chosen AES key and Cyclic Redundancy Check (CRC) is computed across the ciphertext.

The Zigbee forward and reverse proxies are based on the libxbee library (3). The library is connection-oriented, so a two-way communication is

established using 64-bit addressing. The connection is associated with a callback function that simply sends a short response to acknowledge receipt of a non-empty message. Within the active callback thread, the connection's callback function will execute once for each packet received, in the order the packets were received.

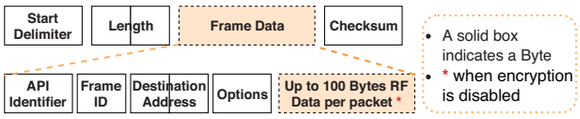

Figure 6: *TX packet (64-bit address) API frame*

By default, the maximum RF packet size is 100 bytes, but when AES encryption is enabled, the maximum packet size is reduced to 95 bytes. A RX packet frame is similar to the TX packet frame shown in Figure 6 except the frame ID and destination address are replaced by source address and RSSI respectively.

4.1.4. Zigbee issues

Two issues arise when pushing HTTP data over Zigbee connections. First, whilst operating in API mode, the Xbee modules do not escape special control sequences: 0x7E (frame delimiter), 0x7D (escape), 0x11 (XON), 0x13 (XOFF) and 0x00 (NULL). If these control sequences, in particular 0x00, appear in the transmitted data it can result in packets being dropped. To combat this, all data in converted into an ASCII representation before transmission. For example, binary data 0x01 will be transmitted as ASCII "01". This allows the complex HTTP headers to be reliably transmitted over Zigbee, but has the downside that twice as much data needs to be transmitted.

Secondly, the Xbee devices have a DataIN (DI) buffer of 202 bytes. To transmit data over the connection, data is written to this buffer over UART, and it is then processed and transmitted. If this buffer is overflowed, then the Xbee module will ignore all bytes received until it sees the next start delimiter, indicating the start of a new API frame. This 202 byte limit is very easy to reach when using HTTP packets due to the header size. This is solved by fragmenting data at the proxy, and reassembling the request when received. The C code for fragmenting can be seen in Listing 1.

Listing 1: C code of fragmentation algorithm

```
void fragment_send(struct xbee_con *
    con, char *in, int len) {

    char buf[(MAX_BYTES*2)+4];

    while (length > 0) {
        if (length > MAX_BYTES) {
            strcpy(buf,"FRAG");
            strncat(buf,in,MAX_BYTES
                *2);
            xbee_conTx(con,NULL,buf);
            in += MAX_BYTES*2;
            length -= MAX_BYTES;
        }else {
            strcpy(buf"ENDD");
            strncat(buf,in,len*2);
            xbee_conTx(con,NULL,buf);
            break;
        }
    }
}
```

4.2. Proxy

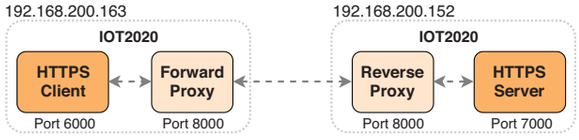

Figure 7: *An overview of the double encryption experimental setup*

In order to simplify the implementation across multiple protocols, a proxy is placed on both the client and the server. The HTTPS client communicates with this proxy, rather than the server. A similar setup is used on the server side, as shown in Figure 7. The proxy acts as a modem, converting the TLS packets into a format usable by the wireless protocol on the client side, and recovering the TLS packets on the server side. This removes the need to modify the HTTPS library for each protocol. Proxies operate in both directions, allowing the client and server to both send and receive. All traffic, including the TLS handshake, is passed through this proxy. A proxy is used for both ethernet and Zibgee connections.

4.3. RESTful HTTPS

We used cURL [1] and the Ulfius library (15) to implement a RESTful HTTP/HTTPS server. cURL is an open source computer software project providing a library (libcurl) and command-line tool (curl) for transferring data using various protocols such as

[1] `https://curl.haxx.se`

HTTP, HTTPS, FTP, SMTP, Telnet, etc. cURL also supports SSL certificates, HTTP POST, HTTP PUT, proxies, HTTP proxy tunneling and more. On the other hand, the Ulfius library is useful for defining RESTful HTTP and HTTPS endpoints. The Ulfius library is based on libcurl (8), so it naturally inherits the framework for handling HTTP and HTTPS connections from libcurl.

The Ulfius framework runs as an asynchronous task in the background of the server. When an Ulfius instance is initialized, a thread is spawned and executed in the background. The thread will listen to the port we defined and dispatch the calls to the callback function linked to the specified endpoint. For the sake of debugging, we have a callback function that reads the request from the port, prints the request out to the standard output stream, and sends a fixed response back to the client. Additionally, we have another callback function that is meant for measuring the round-trip time. It sends a response back to the client immediately after the request is received, keeping the server processing time minimal.

A private key and a self-signed certificate are generated for the server using OpenSSL, using 2048 bit RSA. Then, the server program will initialize the framework and start a RESTful HTTPS server using the RSA key and certificate identified by the file names in the command line arguments. Inversely, the same server program is simply executed without any command line arguments to make the server accept HTTP connections.

5. EVALUATION

5.1. Measurement

Performance is measured using round-trip-time, RTT, which we measure as the time taken to receive a response from the server on sending a request from the client. This includes both layers of encryption, with the start time set before calling the send function on the client, and end time set when the client has received the response, and has the decrypted response data. The cURL command line tool is used as the HTTPS client, as it has built in functionality for benchmarking HTTPS connections. Figure 8 shows a complete HTTPS transaction, including the handshake, and the reported cURL timings. The RTT is calculated by subtracting `time_pretransfer` from `time_total`. In all cases, each experiment is repeated 100 times, with the mean time then presented as the result.

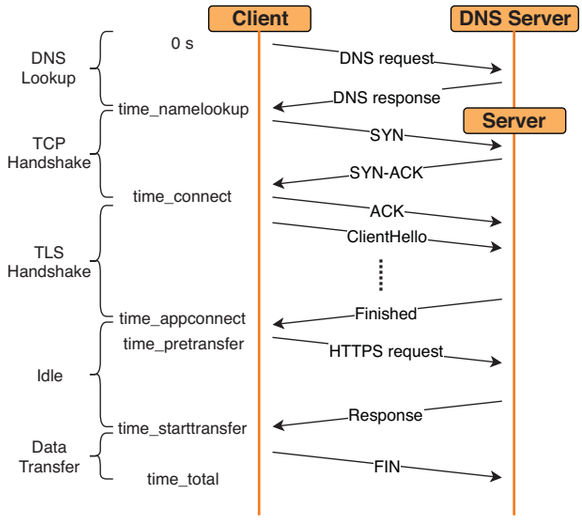

Figure 8: *Timing breakdown of a complete HTTPS transaction*

5.1.1. Payloads

Message payloads are textual data, generated using `/dev/urandom` to create a file random data, encoded as base64.

5.2. Effect of fragmentation on Zigbee

To measure the impact of fragmentation on Zigbee communication, we measure the effect of increasing payload size on the RTT. Data is sent in a raw format, without using HTTP. The server does not respond until all data has been received.

The graph in Figure 9 shows a series of steps, suggesting an increase in RTT as the payload gets fragmented into more packets. In both cases, the RTT increases almost linearly with payload size. The gradient of the graph aligns with the time complexity ($\mathbf{O}(n)$) of our fragmentation algorithm shown in Listing 1. The steps in Figure 9 also hint that the RTT is almost constant when the number of fragments remains the same.

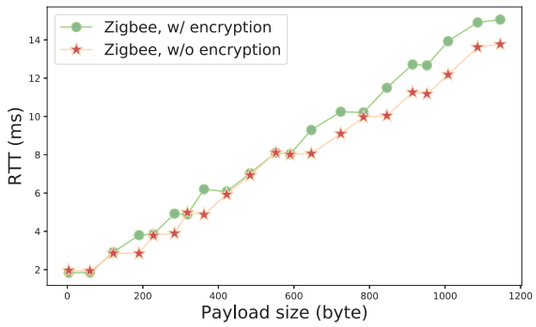

Figure 9: *Mean RTT for Zigbee with respect to payload size*

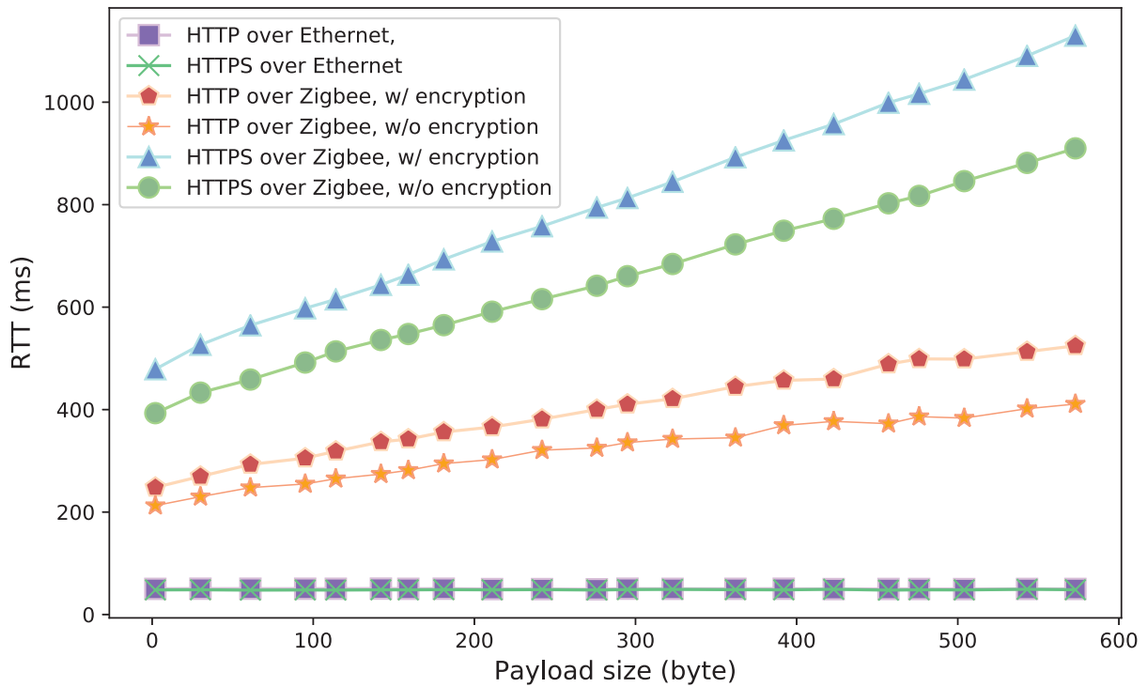

Figure 10: *Mean RTT for different protocols and different encryption configurations, with respect to payload size*

5.3. Overhead of Zigbee encryption

Figure 9 also reveals the overhead of Zigbee encryption on RTT. As can be seen, the added overhead is minimal. As the payload size increases, and more packets are sent due to fragmentation, the overhead increases but does not reach more than 1ms additional delay.

5.4. Impact of double encryption

As can be seen in Figure 10 for ethernet connections, introducing HTTPS instead of HTTP only incurs a mean 1ms increase in RTT, no matter what payload size is used. This is due to there being no fragmentation within ethernet packets, and so the only additional overhead is in the encryption stage which is minimal.

Sending HTTP payloads over Zigbee is substantially slower than over ethernet. Whilst the mean RTT for ethernet is 47ms, a 2 byte HTTP payload over Zigbee without encryption has a mean RTT of 212 ms, increasing to 383 ms for a 500 byte payload. Zigbee encryption introduces a small amount of extra overhead, with a mean RTT of 248ms for a 2 byte payload, increasing to 498ms for a 500 byte payload.

Figure 10 shows that there is a clear performance overhead when sending HTTPS over Zigbee. Without Zigbee encryption, the mean RTT increases from 393ms to 845ms when increasing the payload size from 2 to 500 bytes. When Zigbee encryption is introduced, providing double encryption, this increases to 478ms (2 byte payloads)to 1044ms (500 byte payloads).

5.5. Impact of larger payloads

We also test with larger payloads, up to 10KB in size. Whilst many applications such as wireless sensors are unlikely to require payloads of this size (in particular due to Zigbees low data rate property), we include to measure how heavier applications may perform.

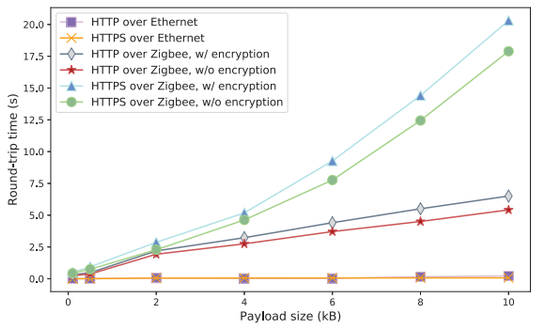

Figure 11: *Mean RTT with respect to payload size (kB)*

As can be seen in Figure 11, RTT increases significantly as payload sizes increase past 2KB.

A 2KB payload over HTTPS and encrypted Zigbee has a mean RTT of 2.86 seconds. This increases to 20.32 seconds for payloads of 10KB. It can also be seen that the additional overhead of HTTPS compared to HTTP over Zigbee also substantially increases as payload sizes increase.

5.6. Discussion

From our results, we find that whilst a RESTful HTTPS protocol over Zigbee is possible, there is a large overhead associated with transmitting HTTP or HTTPS traffic over Zigbee. Transmitting 200 bytes of raw data over Zigbee without encryption results in a RTT of just 3ms, however this increases to 302ms when sending the data by RESTful HTTP over Zigbee. Adding HTTPS does not result in a large amount of extra latency compared to HTTP, indicating that sending HTTP over Zigbee is the primary cause of delay, rather than extra encryption.

We believe that the majority of this delay is in the pre and post processing that needs to be done on the client and server, namely the escaping of special characters, and in particular the fragmentation algorithm presented in Listing 1. As the escaping function described in section 4.1.4 effectively doubles the amount of data that is sent, this could be replaced by a more efficient handling of special characters.

In our experimentation, our Zigbee network consists of only two devices in direct contact. In a real world environment, the Zigbee network could be a far more complex mesh network, requiring multiple hops for communication. This means that cumulative delays across multi-hop routes could be substantial. However, as fragmentation only occurs at the source and destination, the additional delay per hop could be relatively small. This requires further experimentation.

6. CONCLUSION

In this work we aimed to evaluate the feasibility of using RESTful HTTPs over Zigbee networks, in order to provide an additional layer of security in the scenario where the security of the Zigbee network layer is compromised through either a known or presently unknown vulnerability.

There will be use cases where the Zigbee protocol is one of the optimal wireless protocols to use, in part due to its robustness properties and low cost. We show through experimentation that for smaller payloads RESTful HTTPS over Zigbee is a viable option for providing an extra layer of security, as long as a small amount of additional latency is acceptable. For payloads of a few hundred bytes or less, additional latency is measured in the hundreds of milliseconds. Whilst for a safety critical sensor in a industrial automation scenario this additional latency may be unacceptable, there will be use cases where this latency is acceptable. A temperature sensor in a building management system may only report it's value once every 10 seconds, and is not time critical. Non safety critical sensors in a industrial environment, for example those measuring temperature in a water tank or reporting counters on a production line, can also potentially be usable over a higher latency communication. Any use case wherein the transmitted data would fit in a normal Zigbee packet (less than 200 bytes) would perform reasonably with RESTful HTTPS over Zigbee.

We plan to expand on our study to incorporate further wireless protocols, such as WirelessHART. WirelesHART is similar to Zigbee, with a more industrial focus, but has also been shown to have security vulnerabilities (21; 4). We also plan to build a working prototype into the University of Bristol ICS testbed where sensor values can be sent over the RESTFul HTTPS and Zigbee connection, including over a large scale mesh network. .

REFERENCES

[1] C. Alcaraz and J. Lopez. A security analysis for wireless sensor mesh networks in highly critical systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(4):419–428, July 2010.

[2] Z. Alliance. ZigBee Specification. *ZigBee Document 053474r20*, 2012.

[3] Attie. Digi Xbee S1 802.15.4 RF Modules. `http://attie.co.uk/libxbee`, 2012. Accessed: 2019-04-30.

[4] L. Bayou, D. Espes, N. Cuppens-Boulahia, and F. Cuppens. Security analysis of wirelesshart communication scheme. In F. Cuppens, L. Wang, N. Cuppens-Boulahia, N. Tawbi, and J. Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 223–238, Cham, 2017. Springer.

[5] P. Castro, J. Afonso, and J. Afonso. *A Low-Cost ZigBee-Based Wireless Industrial Automation System*, pages 739–749. Springer, 2017.

[6] D. Chen, M. Nixon, and A. Mok. *WirelessHART: Real-Time Mesh Network for Industrial Automation*. Springer, 1st edition, 2010.

[7] S. Coleri Ergen. ZigBee/IEEE 802.15.4 Summary. *UC Berkeley, September*, 10, 2004.

[8] Daniel Stenberg. libcurl - the multiprotocol file transfer library. https://curl.haxx.se/libcurl/, 1997. Accessed: 2019-04-29.

[9] Digi International Inc. Digi Xbee S1 802.15.4 RF Modules. https://www.digi.com/pdf/ds_xbeemultipointmodules.pdf. Accessed: 2019-04-29.

[10] D. Egan. The emergence of zigbee in building automation and industrial control. *Computing Control Engineering Journal*, 16(2):14–19, April 2005.

[11] R. Fielding. REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008. Accessed: 2019-04-25.

[12] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[13] IEEE. IEEE 802.15 WPAN Task Group 4 (TG4). http://www.ieee802.org/15/pub/TG4.html. Accessed: 2019-04-27.

[14] Kudelski Security. Zigbee security basics (part 3). https://research.kudelskisecurity.com/2017/11/21/zigbee-security-basics-part-3/, 2017.

[15] N. Mora. Ulfius. https://github.com/babelouest/ulfius.

[16] C. O'Flynn and Z. Chen. Power analysis attacks against ieee 802.15.4 nodes. In F.-X. Standaert and E. Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 55–70, Cham, 2016. Springer.

[17] C. P. O'Flynn. Message denial and alteration on ieee 802.15.4 low-power radio networks. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–5, Feb 2011.

[18] O. Olawumi, K. Haataja, M. Asikainen, N. Vidgren, and P. Toivanen. Three practical attacks against zigbee security: Attack scenario definitions, practical experiments, countermeasures, and lessons learned. In *2014 14th International Conference on Hybrid Intelligent Systems*, pages 199–206. IEEE, 2014.

[19] G. Ottoy, T. Hamelinckx, B. Preneel, L. De Strycker, and J. Goemaere. On the choice of the appropriate AES data encryption method for ZigBee nodes. *Security and Communication Networks*, 9, 12 2010.

[20] A. R. Raut and L. G. Malik. Zigbee in building industrial control and automation. *International Journal of Wireless Communications and Networking*, 3(2), Dec 2011.

[21] S. Raza, A. Slabbert, T. Voigt, and K. Landerns. Security considerations for the wirelesshart protocol. In *2009 IEEE Conference on Emerging Technologies Factory Automation*, pages 1–8, Sep. 2009.

[22] N. Sastry and D. Wagner. Security considerations for ieee 802.15.4 networks. In *Proceedings of the 3rd ACM Workshop on Wireless Security*, WiSe '04, pages 32–42, New York, NY, USA, 2004. ACM.

[23] M. Sharma, A. Tandon, S. Narayan, and B. Bhushan. Classification and analysis of security attacks in wsns and ieee 802.15.4 standards : A survey. In *2017 3rd International Conference on Advances in Computing,Communication Automation (ICACCA) (Fall)*, pages 1–5, Sep. 2017.

[24] R. Sokullu, O. Dagdeviren, and I. Korkmaz. On the ieee 802.15.4 mac layer attacks: Gts attack. In *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*, pages 673–678, Aug 2008.

[25] R. Sokullu, I. Korkmaz, O. Dagdeviren, A. Mitseva, and N. R. Prasad. An investigation on ieee 802.15.4 mac layer attacks. In *Proc. of WPMC*, volume 41, pages 42–92, 2007.

[26] N. Vidgren, K. Haataja, J. L. Patio-Andres, J. J. Ramrez-Sanchis, and P. Toivanen. Security threats in zigbee-enabled systems: Vulnerability evaluation, practical experiments, countermeasures, and lessons learned. In *2013 46th Hawaii International Conference on System Sciences*, pages 5132–5138, Jan 2013.

[27] J. Waldo, J. Waldo, G. Wyant, G. Wyant, A. Wollrath, A. Wollrath, S. Kendall, and S. Kendall. A Note on Distributed Computing. Technical report, IEEE Micro, 1994.

[28] M. Wilhelm, J. Schmitt, and V. Lenders. Practical Message Manipulation Attacks in IEEE 802.15.4 Wireless Networks. In *MMB DFT 2012 Workshop Proceedings*, pages 29–31, Mar. 2012.

[29] J. Wright. Killerbee: Practical zigbee expliutation framework. https://www.willhackforsushi.com/presentations/toorcon11-wright.pdf, 2011.