# Connectivity, Complexity and the Role of Enhanced Debugging

Glenn I. Matthews
School of Engineering (Electrical and Biomedical)
RMIT University
Melbourne, Australia
glenn.matthews@rmit.edu.au

Trish Messiter; Gokhan Tanyeri
Clarinox Technologies Pty Ltd
Melbourne, Australia
trish@clarinox.com;
gokhan@clarinox.com

*Abstract* — **Over the last three to four years, the adoption of inter-device connectivity and a multitude of other 'smart features' (e.g. adaptive control and device learning) into many traditional embedded products has drastically increased the level of underlying software and hardware complexity. In parallel, product development timelines are rapidly reducing with a corresponding tightening of quality requirements, especially in regulated markets such as the defence, automotive and health sectors. As we race towards a more connected world, with an estimated 26.6 billion devices connected to data networks by the end of 2019, the responsibility lies with designers and developers to ensure that the highly-interconnected systems are robust, reliable and secure. These factors converge to push the need for embedded software engineers to achieve more complex outcomes in less time. To succeed in this challenging environment, engineers need to be armed with adequate tools and efficient processes. A common issue with all embedded systems is the complexity and usefulness of the development tools (both compilers and debuggers). In particular, if a tool is difficult to use then the development process can be greatly hindered rather than assisted. Furthermore, developers are likely to include additional code for debugging purposes which can negatively influence the required timelines and proposed functionality. A need exists for tools which are lightweight, effective and provide complex debugging capability whilst minimising development overhead. This paper explores the role of enhanced debugging in a complex and everchanging field of wireless connectivity. An emphasis is placed on tools which assist in the visualisation of connectivity related data (MQTT, Bluetooth and Wi-Fi) between multiple nodes.**

***Keywords — wireless; connectivity; Bluetooth; Wi-Fi; complexity; security; debugging***

## I. INTRODUCTION

In recent years there has been an exponential increase in the number of connected devices [1]. Due to connectivity requirements, the process for debugging and rigorously testing embedded systems is becoming increasingly more difficult and time consuming. Furthermore, as connectivity requirements increase, vulnerabilities can be introduced into production firmware due to the sheer system complexity. With an increase in system functionality, a requirement also exists for advanced debugging tools in complex, connected, environments.

Whether these devices are connected within the home, or form part of industrial infrastructure, the user places a significant amount of trust that these systems are performing as designed and not running with tampered firmware. A typical use-case scenario is a residential environment with several devices forming a network (whether wired or wireless) that ultimately gain access to internet. As soon as such systems have the ability to reach other connected devices there are substantial issues around device security. As shown with the Mirai Botnet Attack [2], a PC infected with malware will attempt to breach known 'Internet of Things' (IoT) devices with default usernames and passwords. Once compromised, such systems are capable of launching attacks from basic Denial of Service (DoS) up to targeted attempts with the intention of breaching larger security systems. In this case it was due to known username and passwords being shipped with devices, which by default an end-user should be required to update as part of an installation / configuration process.

Security issues are further exacerbated with systems that allow firmware to be remotely deployed. Trends have increased in the number of systems that allow firmware to be updated via 'Over-The-Air (OTA)' techniques [3]. These systems can operate in two basic modes, with either the client or server (manufacturer) requesting (or pushing) an update to a target device. To minimise unwanted tampering with the firmware image, developers can certainly incorporate features such as encryption. However, the complexity of such features are constrained by the processing capacity of the microcontroller receiving the update. Furthermore, it is quite possible that even though the firmware is being sent by a trusted manufacturer, a 'man-in-the-middle' approach could be used to intercept the image, corrupt it for the benefit of an attacker and then redeploy to the remainder of network. This can be quite readily achieved in systems where two-way communication is not performed such as in firmware updates for devices using the terrestrial television network. A networked consumer electronic product, such as a Digital Video Recorder, could be compromised within the confines of a residence and then launch a DoS attack to either all devices within the local network or on a much larger scale [4]. Although the compromised device may no longer be recoverable, all of the other devices on the network must be secure enough to ensure that the damage is minimised.

Comparable examples exist within industrial networks with the potential for more serious consequences.

Although the examples given are relatively simplistic, they highlight the need for robust environment for debugging traffic that is transmitted over wireless interfaces. To date, silicon vendors (such as Marvell, Texas Instruments or Realtek) only supply a firmware image to use on their own devices which then communicate over the chosen interface to the remainder of the embedded hardware which is typically via UART, SDIO or SPI. For the end-user, APIs are not available and hence systems that incorporate hardware from such vendors are reliant upon the device manufacturer. In mission-critical applications such as, for example, control systems managing commercial Unmanned Aerial Vehicles (UAVs) or medical implantable devices such as a pacemaker, the inability to rigorously validate the behaviour of manufacturer firmware and protocol software (and hence the underlying device performance) leaves a substantial gap in the security analysis of a potential product.

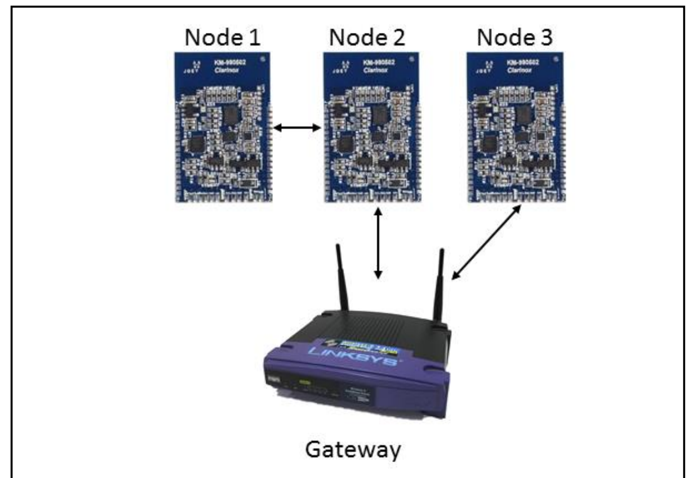
In this paper we discuss and demonstrate the role of debugging tools in the analysis of wireless (Wi-Fi) traffic. An emphasis is placed on the benefits of modern debugging tools versus more traditional techniques. Furthermore, it is demonstrated that modern wireless debugging tools can be used to identify infrequent errors with relative ease, thereby greatly reducing the development time.

## II. BACKGROUND

As previously mentioned, with an increase in the number of connected devices a corresponding increase has occurred in the amount of data being transmitted to a central node. Fig. 1 provides a simple example of three remote nodes communicating back to a central server / gateway. With an increasing trend towards large-scale sensor networks, mesh technologies can be utilized to allow remote nodes (Node 1 in Fig. 1) to communicate back to the central gateway using, for example, Node 2 as a proxy. Furthermore, the deployment of mesh topologies increases the complexity of debugging as nodes may not be directly accessible from the gateway.

Lightweight protocols such as IPv6 over Low-Power Wireless Personal Area Networks (6LowPan), Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport (MQTT) are typically included in low-powered IoT systems to transmit low-rate sensor data [5]. However, as the number of remote nodes increases, the throughput requirement for the central node naturally increases. A further restriction may be based around location and/or device placement where it may not be feasible to use traditional wired networks to transfer large amounts of data for further processing.

To simplify connectivity requirements, it is possible to employ existing technologies such as Bluetooth or Wi-Fi. Although suitable for bulk data transmission, several issues exist around the transmission and debugging of such wireless interfaces. For simplicity a PC can be used to perform the wireless data transmission, however, as their processing power has been increasing at a rapid rate, high-end embedded processors can now replace a PC at a fraction of the cost. A drawback of replacing traditional PC hardware with an embedded processor is that the interface drivers generally need to be redeveloped (including debugging and verification) which




can significantly increase the overall development time.

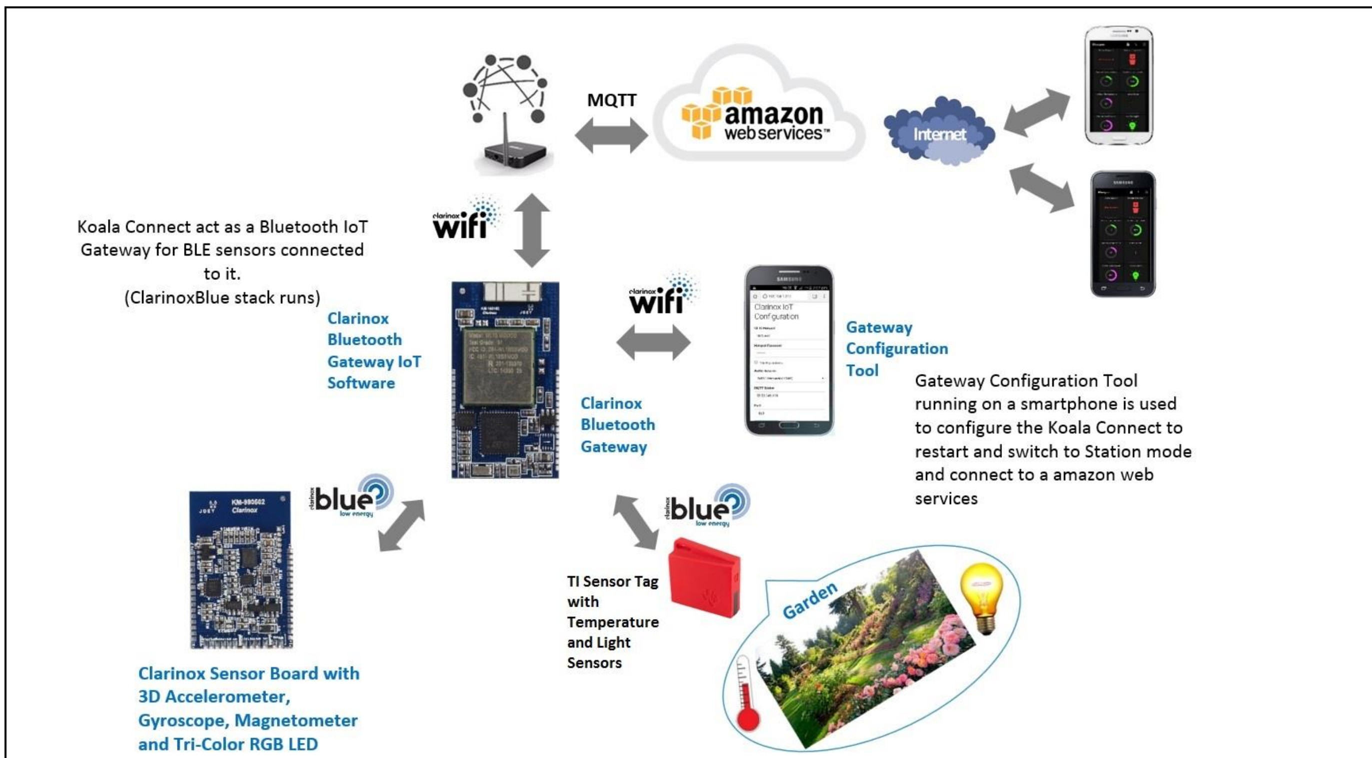
From a developer perspective, the increase in system complexity with reduced production timelines requires tools that allow faults to be diagnosed quickly and efficiently. Traditionally, wireless interfaces have been debugged using tools such as terminal dumps, oscilloscope captures and logic analysers. However, these techniques are generally cumbersome and can often misdiagnose intermittent errors. A review of existing wireless interface debugging tools are further discussed in Sections III and IV.

The issues around preproduction development are similar but with the added limitations of smaller budgets and lower volumes which can easily result in a situation where less vendor support is available, and/or the use of open-source software is necessary. These restrictions can lead to errors (or assumptions) in the underlying drivers being propagated into the preproduction project. Furthermore, in code based on open-source drivers the vulnerabilities are well known and can be exploited.

## III. STANDARD DEBUGGING TOOLS

Whilst the development tools for embedded systems have improved significantly over the last three to five years, traditionally the choices in the realm of wireless systems have been limited. While most commercial systems can be emulated/simulated with relative ease, there are often issues around the simulation of dedicated ICs. Although an idealised model of external hardware can be scripted, this is often not feasible when an actual target is available, particularly when dealing with unexpected inputs. While that is not to say that software simulators are irrelevant, they can often miss core functionality when the physical hardware is available. Hardware level debugging of embedded systems has typically relied on the use of logic analysers (with protocol decode capability) or oscilloscopes. Furthermore, once realised, JTAG can be used to provide debug capability as well as upgrade the underlying firmware.

Fig. 2. Structure of Typical MQTT Environment



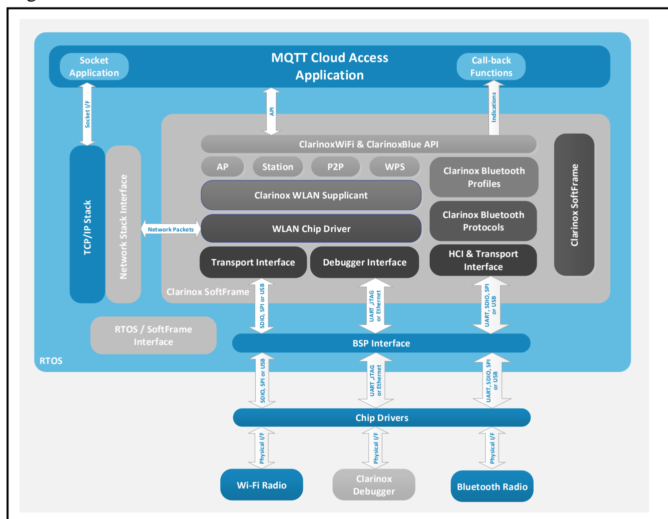

To determine the performance of wireless systems, there are many alternate software simulation platforms. Bluetooth can be simulated using Message Sequence Charts, and tools such as OPNET can be used to simulate physical networks. However, given the often-complex nature of wireless networks it is often difficult to compensate for packet loss, environmental conditions, let alone simulate security breaches. To that end, hardware level tools need to be incorporated as a core part of the development and verification chain. One such technique for analysing wireless systems is the use of a PC platform (with customised packet-sniffing software) however a link then needs to exist between the embedded and the wireless debugging systems. In a mesh network (where systems relay traffic to other nodes) the packet-sniffing approach falls drastically short as traffic between intermediate nodes that are out of range of the central broadcast node (or data sink) can be lost using traditional packet inspection methods. It is these nodes in the distributed network that are the most vulnerable to an attack and hence can compromise the security and stability of the rest of the network.

## IV. CASE STUDY

To illustrate the benefits of enhanced wireless debugging tools, the remainder of this paper will focus on the analysis of a complex scenario in which a gateway has been set up to simultaneously run a Wi-Fi Access Point (AP) and Station (STA) as well as Bluetooth Low Energy. Data is transmitted from a standard sensor (Texas Instruments Sensor Tag) over Bluetooth to the gateway device. An Android device using the MQTT protocol is used to receive the sensor data via the Internet.

Fig. 2 demonstrates the setup for this study. The gateway, comprising a Koala Evaluation Module (EVM) - Clarinox Technologies, routinely transmits data to a broker running on Amazon AWS (Web Services) via a standard access point. In this use-case the Koala EVM runs as in Wi-Fi STA mode and the access point as AP. The Koala EVM simultaneously runs in AP role to enable provisioning and interaction with a standard 4G modem (which connects in STA role). The software block diagram is shown in Fig. 3. This shows the wireless radio, RTOS, ClarinoxSoftFrame, ClarinoxWiFi AP and STA and ClarinoxBlue BLE stack.

Fig. 3. Software Stack



MQTT is used extensively in IoT devices and is based around a publish/subscribe model where sensor nodes routinely transmit data over Wi-Fi to a Broker service. Data from the sensor node consists of topics, to which a client (end-user) can subscribe. In the case of an environmental sensor, the loss (or corruption) of data may be nothing more than an inconvenience. However, in the case of telemetry for a UAV such corruption could result in significant damage, if not worse.

One potential vulnerability in MQTT is that the protocol allows devices to directly communicate with one another (as opposed to only via the central broker). Although the sensor node may be configured to only push data to the broker (in this case, Amazon Web Services), the broker can still send data back to the sensor node. If a vulnerability exists in the processing of data from the broker, then it is possible that the sensor node can be compromised and hence information such as account credentials can be easily obtained.

As previously discussed, with a push to reduce costs, many embedded developers rely on the use of open-source software stacks. While appearing to reduce costs, there is often no simple method in which the entire stack might be examined, both in terms of functionality as well as suitability for a given project. Although many stacks contain additional features that can be disabled with pre-processor directives, this requires developer time to sift through thousands of lines of source code, potentially prolonging development timelines compared to the use of commercial software. Furthermore, if the developers have not written the software themselves (or received the source code from a trusted partner), there is no guarantee that the resulting stack will function correctly, let alone contain sufficient security to minimise the possibility of an attack. It is only via inspection of the data transmitted (whether by wired or wireless means) that the true behaviour of the software stack can be determined and verified. For wired systems, standard debugging techniques (such as logic analysers/protocol analysis) can be used, however the wireless domain is considerably more complex.

In terms of the underlying MQTT protocol there are several open-source clients that can be deployed on embedded hardware. While lightweight versions are commonly available for low-power embedded hardware, such as the ATmega328, these are generally dependent on specific constrained consumer-level networking hardware. The other related issue is that the lightweight versions may not incorporate core security protocols such as SSL / TLS and Quality of Service (QoS) features. In the mainstream market, IBM has produced a C-based client which can be ported to embedded systems. This particular MQTT stack has been validated by Clarinox Technologies.

Before delving into the development tools, there are several important concepts that must be understood when writing applications based on MQTT. As previously mentioned, MQTT operates on a publish / subscribe model. Information that is to be transmitted to another device is published to a broker and the client (or receiving device) subscribes to that particular piece of information which is known as a topic. After a successful connection to the broker, the sensor node subscribes to a variety of topics.

Figures 4 and 5 demonstrates the concept. Here the Koala EVM subscribes to one topic (garden/irrigation) and publishes another known as local/sensor. For simplicity an Android tablet using MQTT Dashboard is used as another node in the network. Both devices are connected to the broker using a Wi-Fi connection via the Internet. Although this example includes two nodes, a typical sensor network may contain thousands of remote devices all attempting to communicate via one gateway to a given broker, which obviously produces significant traffic.

The difficulty in developing firmware for such systems is the ability to accurately intercept the network packets (whether Bluetooth or Wi-Fi) and confirm that the MQTT stack (and indeed the core firmware) is operating as expected. Moreover, if intermediate nodes are present the task becomes increasingly more complex. The increase in complexity and the reduced time-to-market necessitates the development of advanced debugging tools which whilst simple to use, provide a rich feature-set that can be used for both on- and offline debugging.

Traditionally, to debug network traffic several alternative approaches have been used. The most common is to write the packet contents to the debug console and then decode them manually. Although being a time-tested technique, as a manual process it is prone to error and might not highlight the relevant problems with the implementation.

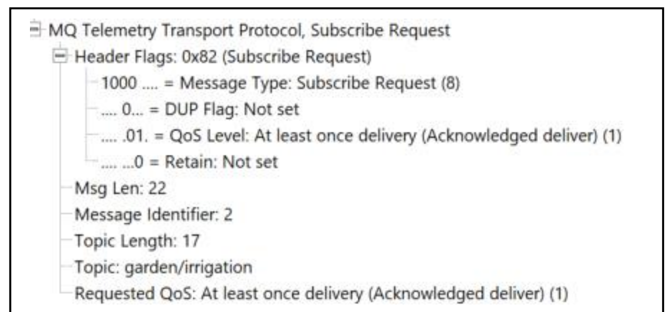
Ideally, a network debugger should be capable of: interfacing directly with the sensor node software stack to decode packets over the Wi-Fi network with limited CPU overhead to capture the MQTT payload. An example of such a tool is the ClariFi debugger. Fig. 4 provides a snapshot of the ClariFi debugger output during a live trace of an MQTT subscribe request (Note that other fields have been minimised for readability).

Fig. 4. MQTT Subscribe Packet Capture

| Trace Buffer | Trace Buffer (hex) |
|---|---|
| ....../.....2Vx. | 08 01 00 00 84 db 2f 16 aa c5 00 12 32 56 78 92 |
| ../............. | 84 db 2f 16 aa c5 00 00 aa aa 03 00 00 00 08 00 |
| E..@......MU.... | 45 00 00 40 00 08 00 00 ff 06 4d 55 0a 00 00 01 |
| 4>0....[....+... | 34 3e 30 1c 9e 97 07 5b 00 00 19 90 2b d8 83 f6 |
| P.Dl..........ga | 50 18 44 6c 88 16 00 00 82 16 00 02 00 11 67 61 |
| rden/irrigation. | 72 64 65 6e 2f 69 72 72 69 67 61 74 69 6f 6e 01 |

In a typical use-case, the trace buffer would be used to manually decode the subscription message. From Fig. 4, the actual MQTT packet begins at byte 73 (0x82) with the preceding data containing the standard TCP/IP fields such as Source / Destination addresses and ports, Time-to-Live and sequence numbers. With the packet successfully captured, it can either be exported using the inbuilt Lua interpreter, or alternatively decoded using the message viewer as per Fig 5.

Fig. 5. MQTT Subscribe Packet Decode

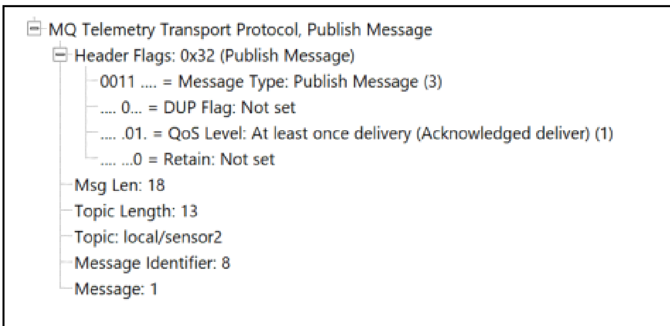


In a similar fashion, the same tools can be used to verify data published to the broker from the sensor-node. In this instance, the value of 0x31 was sent to the broker (and hence the Android Tablet) and displayed. The corresponding packet trace and decode appears in Fig. 6 and 7 respectively. Note that the packet decode (Fig. 7) displays the message payload in plain-text rather than in hexadecimal.

Fig. 6. MQTT Publish Packet Capture

| Trace Buffer | Trace Buffer (hex) |
|---|---|
| ....../.....2Vx. | 08 01 00 00 84 db 2f 16 aa c5 00 12 32 56 78 92 |
| ../............. | 84 db 2f 16 aa c5 00 00 aa aa 03 00 00 00 08 00 |
| E..<......MS.... | 45 00 00 3c 00 0e 00 00 ff 06 4d 53 0a 00 00 01 |
| 4>0....[....+... | 34 3e 30 1c 9e 97 07 5b 00 00 1a 1e 2b d8 84 14 |
| P.DN.v..2...loca | 50 18 44 4e 93 76 00 00 32 12 00 0d 6c 6f 63 61 |
| l/sensor2..1 | 6c 2f 73 65 6e 73 6f 72 32 00 08 31 |

Fig. 7. MQTT Publish Packet Decode




In the previous examples the MQTT communication was successful and contained no errors. The benefits of the debugging platform can be evaluated by the ability to determine the difference between errors and actual network traffic. Using scripting support, Fig 8. depicts the output of the debugger when a packet is unable to reach a destination. The actual packet trace is highlighted to quickly and efficiently indicate to the developer than an error has occurred and needs to be addressed.

Fig. 8. ICMP Packet Transmit Faulture

| Trace Buffer | Trace Buffer (hex) |
|---|---|
| .......Y.v..2Vx. | 08 01 00 00 10 0d 7f 59 1b 76 00 12 32 56 78 92 |
| ...Y.v.......... | 10 0d 7f 59 1b 76 00 00 aa aa 03 00 00 00 08 00 |
| E..8......c..... | 45 00 00 38 00 09 00 00 ff 01 63 93 ac 10 00 07 |
| .......f....E..N | ac 10 00 01 03 03 c3 66 00 00 00 00 45 00 00 4e |
| /.@.@........... | 2f ab 40 00 40 11 b2 cb ac 10 00 01 ac 10 00 07 |
| .....:4. | 04 00 00 89 00 3a 34 d3 |

Whilst the examples given demonstrate the basic functionality of the IBM MQTT stack, such a complex system can hardly be debugged by only analyzing the MQTT events. In real life, such a system will produce many problems related with the underlying connectivity stacks (i.e. Wi-Fi and Bluetooth), Real-Time Operating System (RTOS) related functionality, memory usage problems, TCP/IP stack tuning problems. In addition to these at an early phase of the development, more hardware interface (e.g. device driver related) issues will generally be present. The use of multi-core advanced CPUs introduce more complex driver issues such as caching issues – cache coherency.

Further analysis of the example MQTT gateway application, we see that simultaneous use of Bluetooth and Wi-Fi technologies, in addition to MQTT, will pose many potential problems during the development. When these issues occur one at a time, the traditional tools are sufficient to identify and fix those. However often multiple problems manifest themselves in totally different forms than if they would appear in isolation. Such problems require a more structured approach to identify and remedy.

If the debugging tools do not readily present the information to assist the developers, then there is often the required to delve into a time-consuming exercise to understand the details of the Bluetooth and Wi-Fi technology. As an example, if the Wi-Fi driver is overloading the CPU, causing some of the critical Bluetooth Low Energy sensor reading to be missed, then we are not only looking for a programming logic error, but also a potential tuning problem. In such a case a deep understanding of the protocol is necessary to uncover the fact that packets are missing. An efficient approach is to take the collected logs and perform post processing to programmatically identify the issue. Such post processing of the collected Wi-Fi and Bluetooth protocol logs may assist figuring out when/why the overloading is occurring, significantly more quickly than manual methods. In addition to being faster, such programmatic methods also have the advantage to be repeatable and more suitable for quality procedures.

Essentially debugging can be performed via a number of tools such as the Integrated Development Environment (IDE), terminal dump, oscilloscope, protocol analyser, air sniffer or software specific debugger. Each has its place in the engineer's toolkit. A comparison of these tools is provided in Table I.

Whilst the examples given demonstrate the basic functionality of the IBM MQTT stack, the benefits of advanced debugging tools have been clearly demonstrated. The use of such debugging tools can significantly accelerate the debugging/release cycles by providing developers with the necessary interfaces to quickly, efficiently and concisely verify their firmware images over complex wireless networks.

TABLE I. DEBUGGER CHARACTERISTICS

| **Characteristics / Feature** | **IDE** | **Terminal Dump** | **Oscilloscope** | **Protocol Analyser** | **Air Sniffer** | **ClariFi** |
|---|---|---|---|---|---|---|
| Plain text display | ✓ | ✓ | x | x | ✓ | ✓ |
| Graphical display | x | x | ✓ | ✓ | ✓ | x |
| RTOS event display | ✓ | ✓ | x | x | x | ✓ |
| Easily relatable to other system events | ✓ | x | x | x | x | ✓ |
| Realtime display / Realtime process | x/ x | ✓/x | ✓/x | ✓/✓ | ✓/✓ | ✓/✓ |
| Hardware Price | Med | Low | Med | High | VHigh | Med |
| Engineering Time | Med | VHigh | High | Med | Low | Low |
| Does not mandate use of other tools to convert e.g. Wireshark | x | x | x | ✓ | ✓ | ✓ |

## V. CONCLUSION

In this paper an introduction to the need for advanced debugging tools in complex wireless environments has been given. It has been shown that such tools provide the ability to debug wireless interfaces which are typical of current trends in connected hardware. By providing developers with efficient tools, the time-to-market can be significantly reduced, and robustness and security increased, thereby providing an edge in a highly-competitive consumer market.

## ACKNOWLEDGMENT

This work was partially supported by the Australian Government – Department of Industry, Innovation and Science – Innovation Connections Grant (ICG000408).

## REFERENCES

[1] R. Möller, "Ericsson Mobily Report," Ericsson, Stockholm, 2018, pp.16

[2] C. Kolias, G. Kambourakis, A. Stavrou and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," in Computer, vol. 50, no. 7, pp. 80-84, 2017. doi: 10.1109/MC.2017.201

[3] V. S. Varadharajan, D. S. Onge, C. Guß and G. Beltrame, "Over-the-Air Updates for Robotic Swarms," in IEEE Software, vol. 35, no. 2, pp. 44-50, March/April 2018. doi: 10.1109/MS.2018.111095718

[4] N. Vlajic, D. Zhou and J. Tung, "IoT Cameras and DVRs as DDoS Reflectors: Pros and Cons from Hacker's Perspective," 2018 IEEE International Conference on Industrial Internet (ICII), Seattle, WA, 2018, pp. 181-187. doi: 10.1109/ICII.2018.00035

[5] C. Sharma and D. N. K. Gondhi, "Communication Protocol Stack for Constrained IoT Systems," 2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU), Bhimtal, 2018, pp. 1-6.doi: 10.1109/IoT-SIU.2018.8519904