RAPID7

WHITE PAPER

THE WEAPONIZATION OF CELLULAR-BASED IOT TECHNOLOGY

Deral Heiland - Principal Security Researcher (IoT), Rapid7

Carlota Bindner - Lead Product Security Researcher, Thermo Fisher Scientific

INTRODUCTION

Over the last few years, we have performed focused research around cellular-based technologies used in Internet of Things (IoT) technologies. The goal was to expand our knowledge in the areas around how it works: How we can better understand its usages and ecosystem; how to test the technology from a security perspective; how the cellular technology and its overall IoT ecosystems could be exploited; and how we protect and secure cellular IoT devices and their ecosystem from any identified security issues.

In the first two phases of this research we covered many aspects of cellular technology in IoT, including understanding and interacting with this technology from a hardware perspective and validating and testing the practicality of how those attacks could be implemented.

During that research, we alluded to how an attacker with physical access to these devices could leverage the functionality of cellular modules to carry out attacks against the device's trusted ecosystem (cloud, internet, and private backend network accessible services). We also discussed certain attack scenarios and showed, in detail, how they are carried out against IoT technology and its overall ecosystem.

As cellular connectivity continues to be integrated into devices that interface with critical backend services and private cloud environments, the implicit trust placed in cellular communication paths presents an increasingly attractive target for adversaries. The goal of our research has been to expand our knowledge in the areas around how it works:

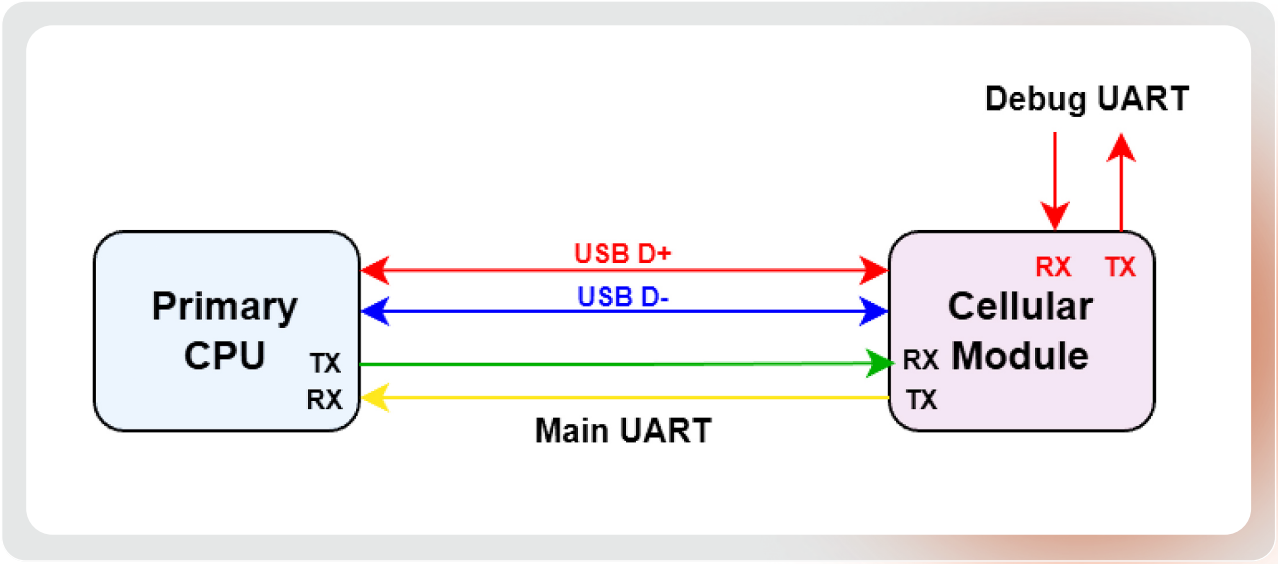
- How we can better understand the technology's usages and ecosystem.
- How to test the technology from a security perspective
- How cellular technology and its overall IoT ecosystems could be exploited

In this phase (weaponization) of our research we demonstrate that without appropriate safeguards at the hardware, device, and network layers, cellular-enabled technologies can be abused to gain unauthorized access, potentially exfiltrate critical data, or pivot into backend network infrastructure. In this research paper we discuss the practicality and processes around how those weaponization attacks would work and how they could be built and implemented.

Previous research: Interchip communication sniffing and attacks

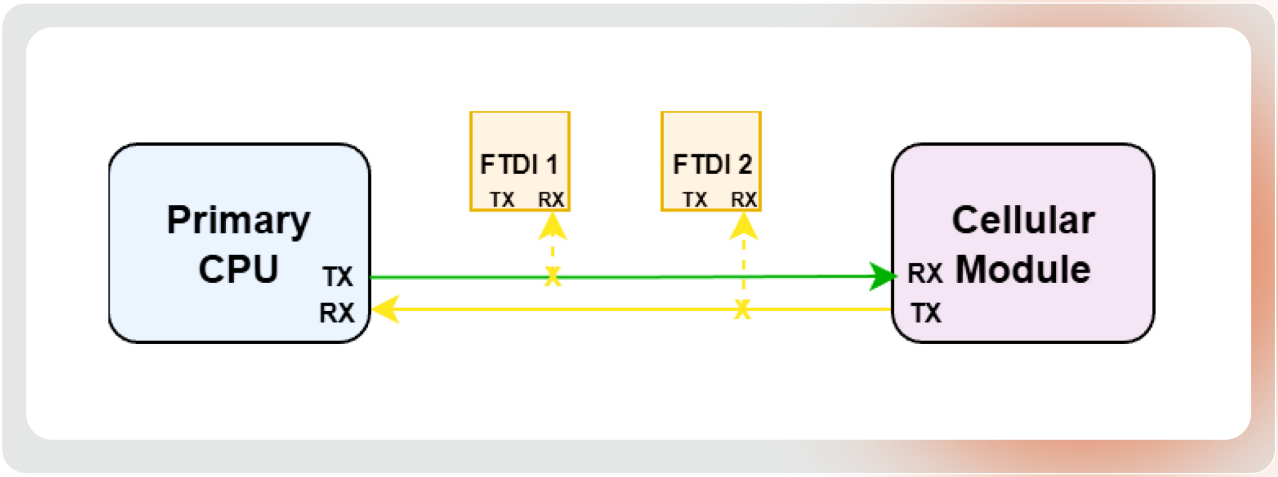
Prior to weaponizing cellular-enabled IoT devices, we examined interchip communication to understand how the cellular module received commands for communicating with external resources. Since this was detailed in our previous paper, "Analysis Of Cellular Based Internet Of Things (IOT) Technology," this section provides a high-level overview intended to give readers the necessary foundation for understanding how internal communication visibility and control facilitated the attacks described later in this paper.

Through circuit board analysis, we identified two primary communication mechanisms between the cellular module and the CPU in cellular-enabled devices: USB and UART. The image below provides a simplified view of these two primary interchip communication pathways.



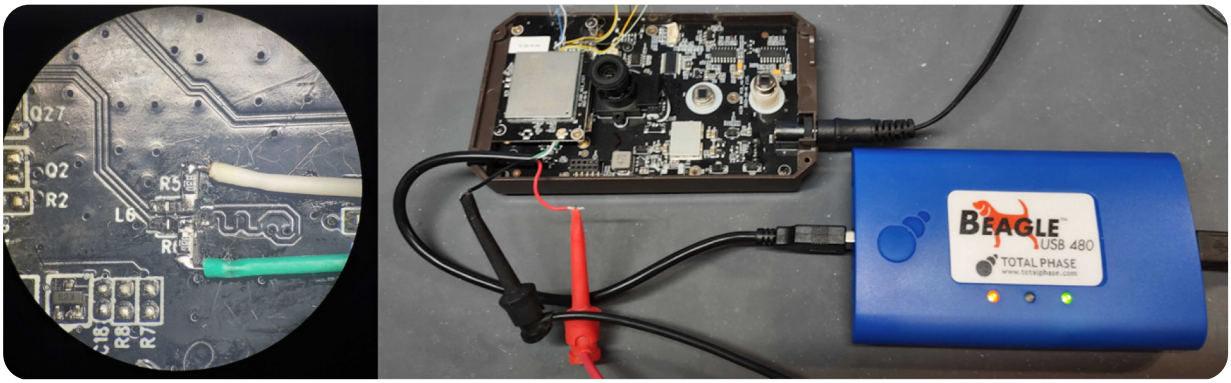
Although USB and UART both serve as interchip communication channels between the host CPU and cellular module, each interface requires distinct tooling and techniques for effective data sniffing. To capture and analyze communication between the cellular module and host CPU, we first identified the relevant PCB traces connecting the two components.

For UART-based communication, we used two FTDI USB-to-serial adapters to observe bidirectional serial data exchange between the components. As shown in the figure below, the RX pin on each FTDI device was connected to the corresponding UART transmit line between the cellular module and the CPU.



To view the captured UART traffic, we used two instances of CoolTerm, a serial terminal application compatible with FTDI devices. It should be noted that in our previous work, we observed that the Main UART on the trail camera, our primary example device, was not used by the CPU to send AT commands to the cellular module. Instead, we only observed an RDY response when the cellular module powered up and a POWERED DOWN response when the device was shut down.

For USB-based communication, we used a Total Phase Beagle USB 480 protocol analyzer with the associated Data Center software to monitor high-speed USB 2.0 traffic between the host CPU and cellular module. In addition to connecting to the D+ and D− lines for USB traffic, we supplied an external power source to provide the 5V VBus required by the Beagle for reliable sniffing. The figure below shows the Beagle connected to the trail camera along with the external 5V power supply used for VBus.



We found that USB was the primary communication pathway between the cellular module and the CPU. In the Total Phase Data Center software we captured AT commands, including authentication data associated with Amazon S3 buckets, as well as the transmission of images and video generated by the camera.

Other previous research: SIM card attacks

In our current research, we did not focus on SIM card attacks. Instead, we targeted systems with the assumption that various security measures, such as IMEI validation checks, eSIM implementation, and SIM PIN configuration, may be in place. Our goal was to leverage the fully functioning embedded IoT hardware, including its trust relationships and internal security. With that said, it's important to first point to previous work by researchers that show how SIM-related vulnerabilities can impact the security ecosystem and trust relationships within an IoT device's ecosystem.

Although the media often focuses on consumer mobile phones when reporting SIM card attacks, cellular-enabled IoT devices are just as vulnerable. In 2019, AdaptiveMobile released a technical paper on SimJacker[1]— a vulnerability found in the S@T Browser, a SIM card technology that, by default, lacked authentication and could be used to execute functions against the SIM card without a user's knowledge. AdaptiveMobile found that Simjacker could be exploited through various attack vectors to extract information, including device location. While AdaptiveMobile's paper focused on consumer cell phones, they did mention S@T browser technology could be in use by IoT devices, depending on the mobile operator.

Additionally, on Mar. 17, 2025, security researchers at Security Explorations disclosed to Kigen, a manufacturer of eSIMs for IoT devices, a vulnerability in the Kigen eUICC card's GSMA TS.48 Generic Test Profile, versions 6.0 and earlier[2]. The vulnerability could enable the installation of unverified applets, and the researchers demonstrated how it could be leveraged to extract the Kigen eUICC identity certification. The certificate compromise demonstrated how the vulnerability posed a risk not just to devices but also to mobile network operators (MNOs), as it could allow unauthorized access, extraction, and manipulation of sensitive mobile network data.

1 https://info.enea.com/Simjacker-Technical-Paper
2 https://security-explorations.com/esim-security.html

For those interested in learning more about security research for SIM cards, the paper "SIMurai: Slicing through the Complexity of SIM Card Security Research" provides valuable insights into SIM card technology while introducing a software platform for security-focused SIM experimentation.[3] Additionally, one of the authors, Tomasz Lisowski, co-presented related work at REverse 2025 in a talk titled "SIMSalabim: Blitz and Tricks with SIM cards,"[4] which examined SIM behavior through the reverse-engineering of SIM interactions within modern phones and the security issues that can arise.

Weaponization attacks and goals

Cellular based IoT devices typically authenticate to share data and receive remote management and control commands. The authentication methods used and what they authenticate come in many forms and may include:

- Cellular network service authentication Access Point Name (APN)
  - Public
  - Private
- Cloud services
- Backend network infrastructure and services

The methods used for authentication to various services for the IoT technology to attach to its ecosystem and function as it was designed for may include the following:

- Secure challenge-response[5]
- IMEI validation
- Password authentication
- Token
- Keys
- M2M Inherent trust

Once authenticated, there is an implied level of trusted access, and in many cases, there may be various shared trusts with other devices. Most off-the-shelf IoT devices connect to cellular services via public APN, which allows access to the internet and arious internet-exposed devices and resources, but some devices connect via private APN, which can allow access to internal network infrastructure or private cloud services. Since we are targeting attacks against the physical hardware and leveraging its own authentication and trust, we take advantage of all internet security mechanisms put in place on the device to protect against unauthorized access. This makes devices utilizing private APN access the highest risk for exploitation using the methods we discuss in this paper.

Hardware modification attacks

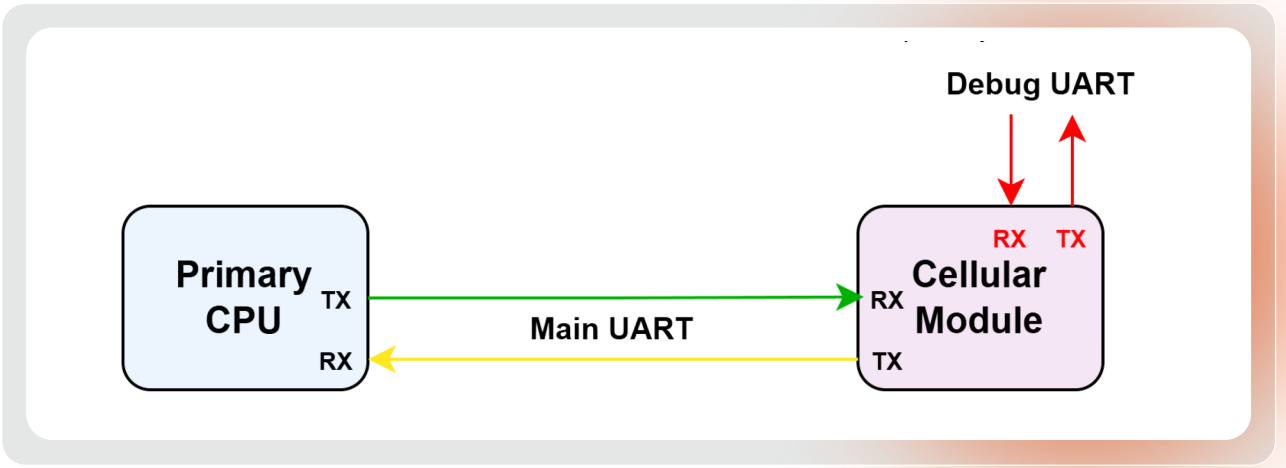
In the following section, we address various hardware modification attacks aimed at taking control of the device's internal communication channels. These modifications allow us to leverage the device's inherent security trust with external systems such as cloud, web, and internet accessible services, along with backend network connectivity for select devices, which establish VPN connectivity to internal networks. In general, every cellular module we encountered during our research allowed for command and data communication to the cellular module via the main UART and USB.

3 https://www.usenix.org/system/files/usenixsecurity24-lisowski.pdf
4 https://www.youtube.com/watch?v=2ou_MxSmiVo&t=5s
5 https://en.wikipedia.org/wiki/SIM_card

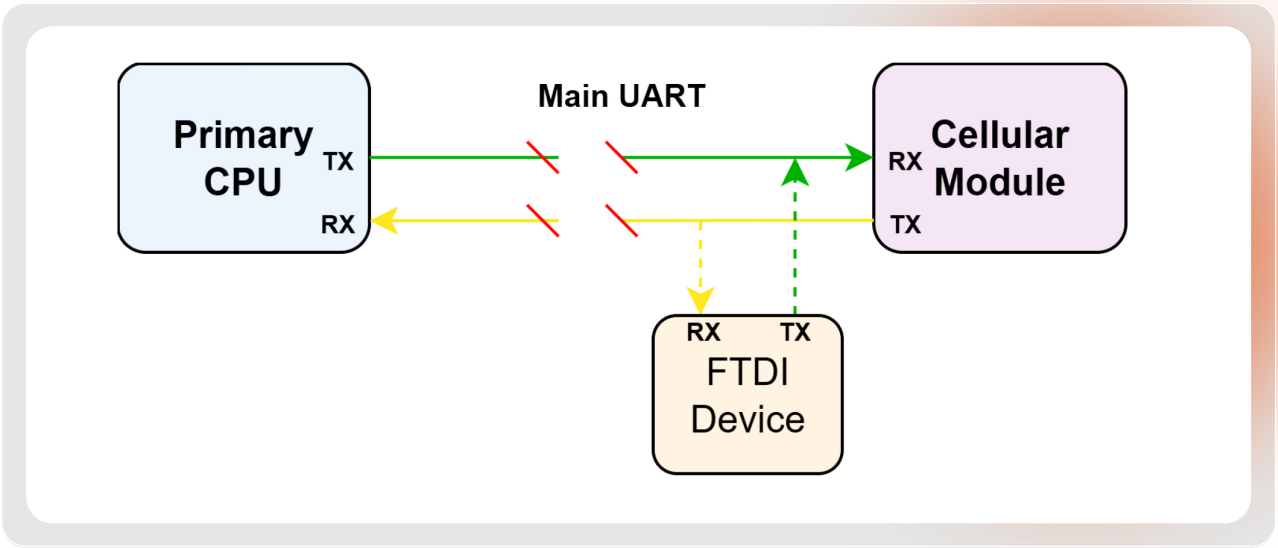
Typically, only one of these services was connected to the device CPU, and in many cases, the unused one was not connected. Using physical attacks on the devices under test, we were able to establish a connection to the unused command and data line, whether it was USB or UART, to control the device. Also, we have been able to leverage the active command and data lines, whether USB or UART, either by man-in-the-middle (MitM) attacks of the communication or completely hijacking the communication line after the device's full authentication to back-end services was completed.

UART

Cellular modules are usually designed with two UART interfaces: a debug UART and a main UART. The debug UART interface serves as a diagnostic and development interface for the cellular module, while the main UART is one of the two interfaces the cellular module supports for AT command communication. When debug UART is enabled on the cellular module, we can obtain basic device information during its initial boot sequence and possibly access the debug console; however, we focused on the cellular module's main UART and its data communication with the primary CPU.



We extended on earlier work presented in Leveraging Inter-chip Communication Analysis for Examining End-to-End Security within IoT Technology to analyze interchip communication on the main UART between the cellular module and the CPU. While we initially tapped into the main UART using two FTDI serial-to-USB devices, each connected to one of the TX/RX lines between the cellular module and primary CPU, this only allowed us to observe the command and data communication. To issue AT commands to the cellular module, we cut the main UART connection between the cellular module and the CPU and connected a single FTDI serial-to-USB device to the cellular module's main UART.



This configuration effectively allowed us to intervene in the primary CPU's communication, which normally sends instructions to the cellular module about data transmission, and gain the ability to issue our own commands.

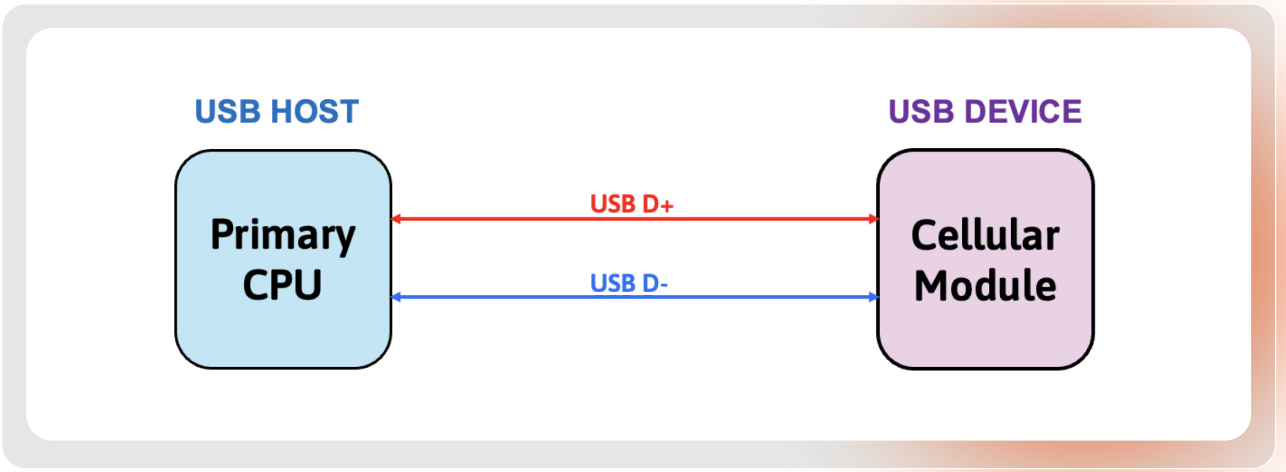
USB

As previously discussed, cellular modules support two types of command and data communication connections to the cellular module, UART and USB. As part of our research, we wanted to understand how we could leverage USB access to the cellular module within the device's current circuit design to allow us control and forward data through the cellular module.

To accomplish this, we considered various options. One of those options was covered in our previous research paper, where we discussed tapping into the USB circuit using a Beagle USB 480[6] sniffer to capture and decode the traffic. This was very effective for understanding the device's communication, ecosystem interactions, trust relationships, and cellular USB configuration, but did not allow us an effective way to inject command and control data into the architecture. To do this, we considered various ways to modify the USB on the circuit board that would allow us to potentially inject data or hijack the full communication channel.

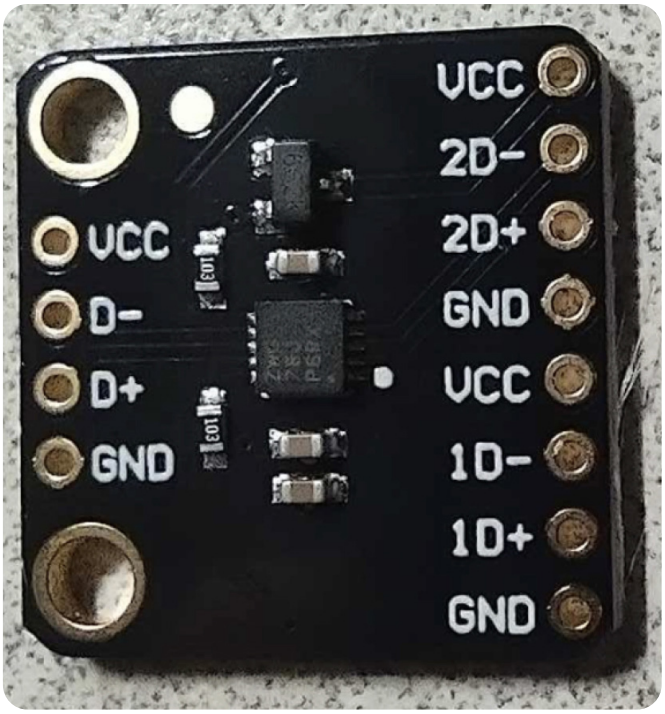
When considering USB communication, the USB 2.0 standard[7], typically encountered on NB-IOT[8] and LTE-M[9] cellular modules, is most often found deployed within IoT technology. USB 2.0 has two types of communication nodes: Host and Device. Within a USB communication bus, there can be only one Host; the Host device controls the communication channel, which is the primary CPU of the IoT device. Also, there can be many Devices on a USB 2.0 bus, although this is typically limited by bandwidth and power requirements. On a cellular-based IoT device, we find only one USB 2.0 Device and that would be the cellular module.

6 https://www.totalphase.com/solutions/apps/usb-analyzer-guide/
7 https://www.usb.org/document-library/usb-20-specification
8 https://en.wikipedia.org/wiki/Narrowband_IoT
9 https://en.wikipedia.org/wiki/LTE-M



Since only one Host and one Device are typically deployed with the IoT technology, we needed to consider an attack path that would replace the Host device and allow us to control the cellular module. As part of this line of reasoning, we assumed that the attack should occur only after the primary CPU and cellular module had completed security negotiation and established trusted access, at which point we would attempt to hijack the communication by replacing the USB host (primary CPU).

To accomplish this style of attack, we conducted a search for electronic components that would allow for multiplexing of USB communications. During this search, we located the Texas Instruments TS3USB221E,[10] which is a USB 2.0 1:2 multiplexer. This electronic integrated chip (IC) component was designed to typically allow switching between devices, but our plan was to use it to switch between the Host in a 2:1 style configuration. Through further research, we located an assembled module[11] hosting the TS3USB221E multiplexer IC. This module contained all the needed circuit and support components, removing our need to design a circuit board module.



10 https://www.ti.com/lit/ug/scdu001a/scdu001a.pdf?ts=1761507412321
11 https://www.amazon.com/Rakstore-TS3USB221-High-Speed-480Mbps-Demultiplexer/dp/B099NPVWP3/

So, with this module in hand, our next step was to splice this module into the IoT device's internal electronic circuit without disrupting the normal USB communication flow. Before doing this, we focused our research on understanding the electronic specification of USB communication. In that phase of research, we identified several areas we needed to focus on to avoid degrading the USB communications.

Typical design specifications of printed circuit boards (PCB) having USB communication capabilities require a differential impedance of 90 Ω between D+ and D-. As typical hardware hackers, we did not have advanced tooling for measuring these impedances, but we still must consider what modification we make to the overall design of the device during testing and research so that it does not impact required impedance outside of its 10-15% tolerance. During this process, we needed to consider the following items, which could seriously impact this impedance specification and include:

- Trace skew
- Trace branches
- Ground plane continuity

Trace skew: when splicing new circuits, such as the above multiplexer board into the IoT devices circuit, you must avoid a length differential between D+ and D-. This skew,[12] if miscalculated, can have a serious impact on the device's ability to continue functioning at High Speed (HS) 480mhz. On a positive note, the USB 2.0 HS standard is very tolerant. Although detailed skew limit specifications[13] [14] for circuit board designs can be vague, only skew limits for USB cables which are 100 pico seconds (ps) were identified.

Using the skew limit of 100 pico seconds as a baseline, we can then reduce that limit to half (50 ps) as best practice for circuit design and apply standard material propagation delays[15] (180–200 ps/inch on FR-4). FR-4 is the most common base material used to make printed circuit boards (PCBs) we then apply that to the following max differential skew limit formula:[16]

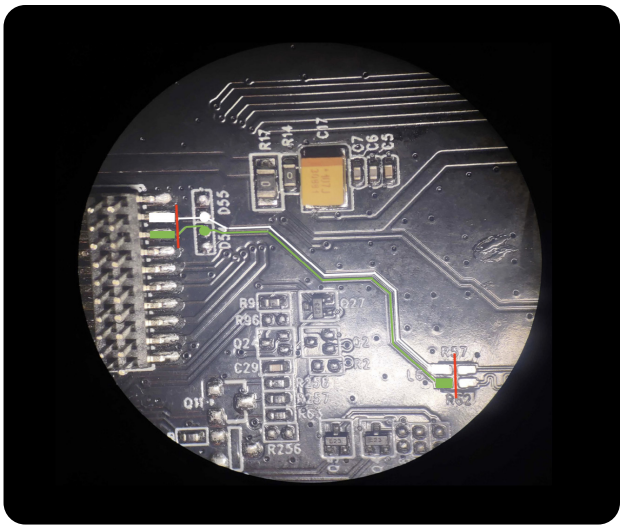
$$\left| L_{D+} - L_{D-} \right| \leq \frac{T_{\text{skew,max}}}{T_{\text{pd}}}$$

We came up with a max skew limit of .28 inches, as shown below, which is large in the world of PCBs, making it simple, with a little effort, to modify the USB circuit to add a multiplexer into the circuit and stay within those limits.

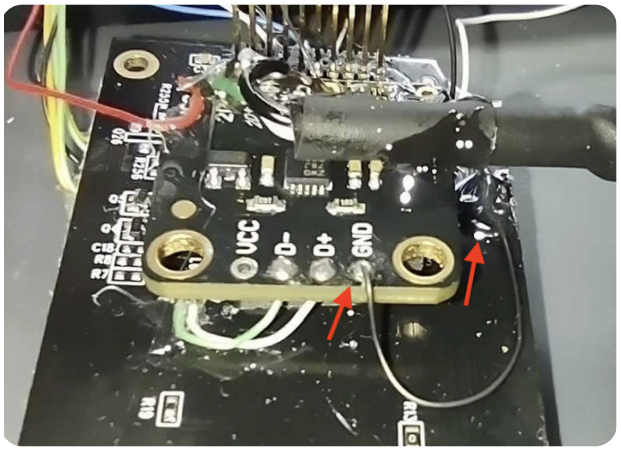
$$\frac{\left| L_{D+} - L_{D} \right|}{} = \frac{50\,\text{ps}}{180\,\text{ps}} = 0.28\,\text{in}$$

12 High-Speed Layout Guidelines for Signal Conditioners and USB Hubs - https://www.ti.com/lit/an/slla414/slla414.pdf
13 USB 2.0 Specification § 7.1.3 Cable Skew - https://www.usb.org/document-library/usb-20-specification
14 Silicon Labs AN0046 — USB Hardware Design Guidelines - https://www.silabs.com/documents/public/application-notes/an0046-efm32-usb-hardware-design-guidelines.pdf
15 IPC-2141A Design Guide for High-Speed Controlled Impedance Circuit Boards - https://www.electronics.org/TOC/IPC-2141A.pdf
16 A Handbook of Black Magic (H. Johnson, Prentice Hall 1993) - https://mrce.in/ebooks/High%20Speed%20Digital%20Design%20Handbook%20of%20Black%20Magic.pdf

Trace branches:[17] When modifying a circuit to reroute communication, it is important to avoid abandoning signal branches that are no longer in use. If branches are left in place, they will impact the signal by causing signal reflections and degrading the required 90 Ω differential impedance. Before attaching the multiplexer in circuit, we made sure all unused signal branches were removed. An example of this being done on our research project device is shown on the right, where we disconnected on both ends (marked in red), the PCB USB circuit branch (outlined in white and green) that was no longer being used.



Ground plane continuity:[18] Differential USB signals (D+ and D-) require a consistent return path. Within the USB pair (D+ / D–) both sides independently create an opposing current that returns through the ground plane. If that ground plane is split, or not evenly connected under the signal path, the return current is forced to detour which can lead to adding inductance, impedance, and noise to the signal path, degrading the communication, which can prevent the USB 2.0 from properly communicating at HS.



To prevent this, we installed the multiplexer board and attached the board's ground plane to the main ground plane of the IoT device's PCB at a single location, as shown below, ensuring we didn't split or disrupt the ground plane's continuity.

Attaching a ground plane to the multiplexer board in a single location worked well in our example. Although if you do experience signal degradation, the possible cause could be related to material propagation within the overall ground plane. In that case, I would suggest attaching the ground plane in multiple locations from the multiplexer board to the IoT device ground plane. This would help lower overall ground impedance and control the return-current path, improving signal propagation within the PCB circuit.

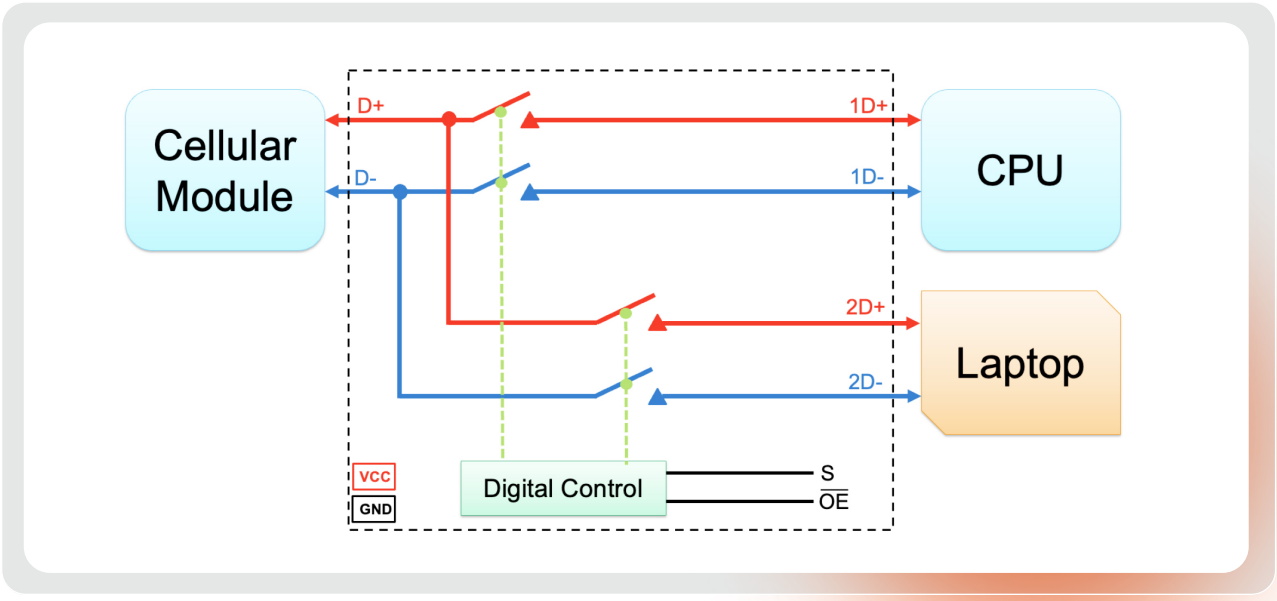
Using the above electronic guidelines, we successfully spliced the multiplexer circuit board into the cellular-based IoT hardware under test. We successfully validated it by using the following TS3USB221E[19] truth table, to initially enable the Multiplexer to communicate between CPU and the Cellular Module via 1D.

17 Texas Instruments, "High-Speed Layout Guidelines for Signal Conditioners and USB Hubs" section 4.7- https://www.ti.com/lit/an/slla414/slla414.pdf?utm_source=chatgpt.com&ts=1763060577857&ref_url=https%253A%252F%252Fchatgpt.com%252F
18 Everything You Need to Know About Stitching Vias - https://resources.altium.com/p/everything-you-need-know-about-stitching-vias
19 https://www.ti.com/lit/ug/scdu001a/scdu001a.pdf?ts=1761507412321

S	OE	FUNCTION
N/A	High	Disabled Multiplexer
N/A	Low	Enabled Multiplexer
Low	Low	D=1D
High	Low	D=2D

Note: Low is achieved by holding the pin to ground; High is achieved by not connecting and leaving it in a floating condition.

Once our initial validation test showed that the multiplexer did not impede the IoT test device from operating in its normal state, we next connected a laptop computer to the 2D USB connection for testing. Below is a pseudo-schematic showing how the multiplexer is connected within the circuit, electrically.



Once the laptop was connected to the multiplexer, we set OE Low to enable the multiplexer and initially set S Low before powering up the IoT test unit. Once the IoT device was up and had completed authentication with APN cellular services and associated backend and cloud systems, we raised S to the High state, switching the Host input on Multiplexer to 2D.

Immediately, the Laptop detected the device's cellular module, Quectel EG91-NAXD, as shown below, and negotiated the connection. The laptop was assigned an IP address by the cellular modules, which was configured to support Ethernet Control Mode (ECM). This allowed us to route all communication from the laptop through the IoT test device to cellular services and backend validated systems.

```
| +-o USB2.1 Hub@00100000  <class IOUSBHostDevice, id 0x10000e98$
|   +-o Tigard V1.1@00140000  <class IOUSBHostDevice, id 0x10001$
      EG91-NAXD@00120000  <class IOUSBHostDevice, id 0x1000150$
+-o AppleT8122USBXHCI@02000000  <class AppleT8122USBXHCI, id 0x1$
| +-o USB Optical Mouse@02100000  <class IOUSBHostDevice. id 0x1$
```

Attack tool

In general, both NB-IoT and LTE-M cellular modules support a number of communication services accessed and controlled using AT commands, including HTTP/HTTPS, TCP/UDP sockets, MQTT, and PPTP. Once we completed building our knowledge and understanding of these cellular module AT commands, along with interaction with the hardware using interchip communications and hardware hacking methods to hijack both UART and USB on the PCB, we focused on building out proof of concept (POC) tools to leverage these capabilities to exploit the device's trust relationship's access to backend services, cloud, and networks.

The first area we will focus on is around leveraging TCP/UDP sockets on cellular modules. In general, many LTE-M cellular modules support some form of raw socket control, which allows for interaction and control of the module to enable the IoT device to communicate directly with services available through the cellular connection. For a small sampling, the following AT socket commands are available on the following listed cellular module brands:

- Quectel[20]
  - +QIOPEN: Open Socket
  - +QISEND: Send data through socket
  - +QIURC: Unsolicited Result Code – Configure incoming data processing and data notification
  - +QIRD: Read incoming data from buffer
  - +QICLOSE: Close socket
- Telit Cinterion[21]
  - +CIPSTART: Open Socket
  - +CIPSEND: Send data through socket
  - +CNMI: Unsolicited Result Code – Configure incoming data processing and data notification
  - #SRECV: Read incoming data from buffer
  - +CIPCLOSE: Close socket
- U-blox[22]
  - +USOCR: Open Socket
  - +USOWR: Send data through socket
  - +USORD: Read data from socket
  - +USOCL: Close socket

TCP port scanner:

The first proof of concept (POC) code we developed using AT over serial was a rudimentary TCP port scanner, which we constructed using the standard Quectel socket commands +QIOPEN and +QICLOSE along with a couple of basic error response codes of:

- 0 operation successful = Port Open
- 566 Socket connection failed = Port Closed

20 https://www.quectel.com/download-zone/
21 https://www.telit.com/els62-download-zone/
22 https://cdn.lantronix.com/wp-content/uploads/pdf/u-blox-CEL_ATCommands_UBX-13002752.pdf

The following Python main function for loop section incorporates these socket commands and error code responses to conduct port open validation against targeted IP addresses:

```
# Main: builds IP/port lists, opens serial, issues QIOPEN, interprets response
codes, and print results
    for ip in ips:
        seen_valid = False
        for port in ports:
            try:
                port_int = int(port)
            except ValueError:
                print(f"Skipping invalid port: {port}")
                continue
            # Attempt connection
            resp = send_at_command(ser,
                                   f'AT+QIOPEN=1,0,"TCP","{ip}",{port_int},0,0')
            send_at_command(ser, 'AT+QICLOSE=0,10')
            # Parse code2
            m = re.search(r'\+QIOPEN:\s*\d+,(\d+)', resp)
            code2 = m.group(1) if m else None
            # Evaluate
            if code2 == '0':
                print(f"\033[92m{ip}:{port_int} - OPEN\033[0m")
                seen_valid = True
            elif code2 == '566':
                print(f"\033[33m{ip}:{port_int} - CLOSED\033[0m")
                seen_valid = True
            else:
                # if no valid seen yet, tag host down
                if not seen_valid:
                    print(f"\033[91m{ip} - DOWN\033[0m")
                    break
                # else just unknown for this port
                print(f"\033[93m{ip}:{port_int} - UNKNOWN\033[0m")
            time.sleep(1)
        # next IP
    ser.close()
```

We found during testing that the 0 and 566 error responses were all that was needed under normal conditions for validating open ports. Any other responses returned and/or timeouts were typically indications that no host was alive at that IP address. We used that condition to help avoid unnecessary port checks for an invalid IP address. An example output of our PoC tool is shown below:

```
DEMO: ./CatScan.py -IC 45.33.32.128/29 -p 21,443,22,80,8000 --serialport /dev/ttyUSB0
45.33.32.129 - DOWN
45.33.32.130 - DOWN
45.33.32.131:21 - CLOSED
45.33.32.131:443 - OPEN
45.33.32.131:22 - OPEN
45.33.32.131:80 - OPEN
45.33.32.131:8000 - CLOSED
45.33.32.132 - DOWN
45.33.32.133 - DOWN
45.33.32.134 - DOWN
```

The more detailed list of Quectel's error responses for TCP/IP AT commands can be leveraged to build and manage more robust applications and to troubleshoot TCP/IP communication via the AT commands. The table below shows the full error code response list for Quectel TCP/IP AT commands:

ERROR CODE	MEANING
0	Operation successful
550	Unknown error
551	Operation blocked
552	Invalid parameters
553	Memory not enough
554	Create socket failed
555	Operation not supported
556	Socket bind failed
557	Socket listen failed
558	Socket write failed
559	Socket read failed
560	Socket accept failed
561	Open PDP context failed
562	Close PDP context failed
563	Socket identity has been used
564	DNS busy
565	DNS parse failed
566	Socket connect failed
567	Socket has been closed
568	Operation busy
569	Operation timeout
570	PDP context broken down
571	Cancel send
572	Operation not allowed
573	APN not configured
574	Port busy

HTTP cloud enumeration:

Since many IoT devices rely on cloud resources, we created a basic AWS S3 bucket scanner that leverages the cellular module's AT commands supporting HTTP communication. We attempted to model the tool off existing AWS S3 bucket scanners to ensure functionality. The main function begins with a help menu that allows a user to configure the UART interface, the S3 endpoint, a wordlist, a bucket name, and additional arguments, such as verbosity.

```
def main():
    if len(sys.argv) == 1:
        print("""
Usage: python3 s_3_enum_via_at.py --bucketnames <bucket or file> --wordlist
<wordlist file> [options]

Example:
  python3 s_3_enum_via_at.py \
    --bucketnames bucketnames.txt \
    --wordlist wordlist.txt \
    --extensions txt json html \
    --s3-endpoint s3.us-east-1.amazonaws.com \
    --serial-port /dev/ttyUSB0 \
    --assume-on

This probes URLs like:
  https://<bucket>.s3.us-east-1.amazonaws.com/<word>.<ext>
""")
        sys.exit(1)

    parser = argparse.ArgumentParser(description="S3 Bucket Enumerator via
Quectel AT HTTP")
    parser.add_argument("--bucketnames", required=True, help="Single bucket or
file with bucket names")
    parser.add_argument("--wordlist", required=True, help="List of words to
try")
    parser.add_argument("--extensions", nargs="+", default=["txt"], help="File
extensions to append (e.g. txt json html)")
    parser.add_argument("--s3-endpoint", default="s3.amazonaws.com", help="S3
endpoint (default: s3.amazonaws.com)")
    parser.add_argument("--serial-port", default="/dev/ttyUSB0", help="Modem
serial port")
    parser.add_argument("--baudrate", type=int, default=115200, help="Baud
rate")
    parser.add_argument("--assume-on", action="store_true", help="Skip RDY
wait")
    parser.add_argument("--verbose", action="store_true", help="Verbose
output")
    args = parser.parse_args()
```

The script's core functionality relies on Quectel HTTP AT commands to issue HTTP requests and retrieve responses. This process uses three commands in sequence:

- AT+QHTTPURL configures the target URL by specifying the URL length and the maximum time allowed to input the URL after the modem responds with "CONNECT."
- AT+QHTTPGET sends a GET request to the previously configured URL. It supports parameters for read timeout (to_read_time), server response wait time (wait_time), and the maximum download size (data_size).
- AT+QHTTPREAD retrieves the HTTP response data returned by the server and accepts a "wait_time" value in seconds.

In the script, the AT+QHTTPURL command is programmatically provided with the URL length and configured with a static 30-second input timeout. After receiving the initial “CONNECT” response from the modem, the script sends the URL, which is constructed from the user-supplied bucket name, S3 endpoint, wordlist, and optional extension modifier.

The script then issues the AT+QHTTPGET command to the cellular module to send a GET request, with a maximum read timeout of 60 seconds. Finally, the response is read using AT+QHTTPREAD, with a 30-second wait time. While the timeout values in the script are statically defined, all three AT commands support configurable timeout ranges in seconds and can be manually adjusted depending on network conditions or user requirements.

```
def https_get(self, url):
    print(f"[MODEM] HTTPS GET: {url}")
    resp = self.send_at(f'AT+QHTTPURL={len(url)},30', wait="CONNECT")
    if "CONNECT" not in resp:
        raise Exception(f"Failed at QHTTPURL: {resp}")
    self.ser.write((url + '\x1A').encode())
    time.sleep(2)
    self.send_at("AT+QHTTPGET=60")
    buffer = ""
    start = time.time()
    while time.time() - start < 30:
        if self.ser.in_waiting:
            line = self.ser.readline().decode(errors="ignore").strip()
            buffer += line + "\n"
            if self.verbose:
                print(f"[MODEM] << {line}")
            if "+QHTTPGET:" in line:
                break
        time.sleep(0.1)

    code_match = re.search(r'\+QHTTPGET: 0,(\d+)', buffer)
    if not code_match:
        raise Exception(f"No valid +QHTTPGET response: {buffer.strip()}")

    code = int(code_match.group(1))
    body = self.send_at("AT+QHTTPREAD=30", wait="OK", timeout=10)
    if self.verbose:
        print(f"[MODEM] Response Body (truncated):\n{body[:500]}")
    return code, body
```

Finally, the script uses the HTTP response code obtained from the AT+QHTTPREAD command to determine whether the resource exists. If an HTTP 200 or 404 is not returned, the response code is printed for the user, so that additional analysis can be performed.

```
if status == 200:
    print(color_text(f"SUCCESS {url} → HTTP {status}", "32")) #green
elif status == 404:
    print(color_text(f"NOT FOUND {url} → HTTP {status}", "33"))#yellow
else:
    print(color_text(f"RESPONSE {url} → HTTP {status}", "36"))#cyan
```

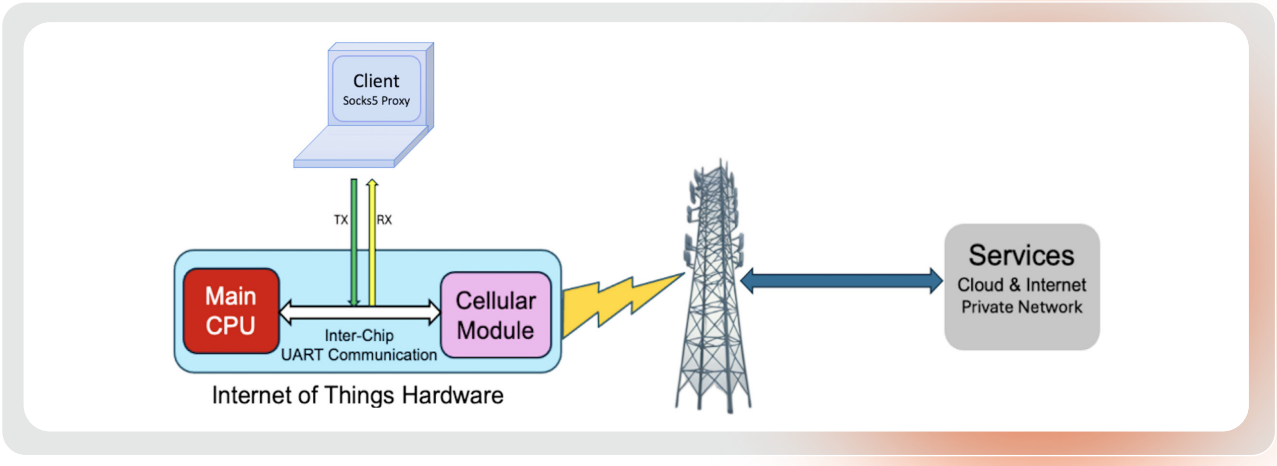
The image below shows the execution for the S3 bucket scanner and the first request, with verbose output displaying commands and responses from the cellular module on the IoT device.

```
phoenix CellScripts % python3 CellS3Enum.py --bucketnames research-cellbucket1 --wor
dlist wordlist.txt --extensions txt --s3-endpoint s3.us-east-1.amazonaws.com --seria
l-port /dev/tty.usbserial-A5069RR4 --assume-on --verbose
[MODEM] HTTPS GET: https://research-cellbucket1.s3.us-east-1.amazonaws.com/flag5.txt
[MODEM] >> AT+QHTTPURL=65,30
[MODEM] << AT+QHTTPURL=65,30
[MODEM] << CONNECT
[MODEM] >> AT+QHTTPGET=60
[MODEM] <<
[MODEM] << OK
[MODEM] << AT+QHTTPGET=60
[MODEM] << OK
[MODEM] <<
[MODEM] << +QHTTPGET: 0,404
[MODEM] >> AT+QHTTPREAD=30
[MODEM] << AT+QHTTPREAD=30
[MODEM] << CONNECT
[MODEM] << <?xml version="1.0" encoding="UTF-8"?>
[MODEM] << <Error><Code>NoSuchKey</Code><Message>The specified key does not exist.</
Message><Key>flag5.txt</Key><RequestId>GVK1Q1H4H3G4T0RT</RequestId><HostId>mcng0p2nc
OSn2zdeOL9ZpQSH29p9WXaFrqsimS7/TK+xwdRJvDR9fys+i76KU6GdVuU8ok5IaCwRnyixRVi8JcP8Uj4jI
RP3Wq9SmaAuWBQ=</HostId></Error>
```

As this script uses Quectel's HTTP AT commands for creating requests, it is limited to a single HTTP transaction at a time because the commands do not provide identifiers for requests or connections. While this limitation could be addressed by refactoring the script to use QIOPEN, as demonstrated by the SOCKS5 proxy found in the next section, we kept the current implementation to demonstrate the constraints that may be imposed by AT commands. This limitation highlights the importance of understanding the capabilities and constraints of AT commands during tool development.

SOCKS5 proxy:

For our next project, we worked on developing a SOCKS5 proxy written in Python that would allow us to route communication from a client device through UART communication to a Quectel EG91-NAX cellular module on our IoT test device. In the following example, we initially leveraged a single socket connection which allowed only simple applications to communicate through the proxy to pivot communication through the IoT device under examination.



By using the Main UART communication on the cellular module, we were able to route application network traffic from our client machine without interrupting the IoT device's normal operation. This is because normal device communication, Ethernet, was handled by the Main CPU via USB to the cellular module leveraging Ethernet Control Mode (ECM).

To accomplish our POC task, we first created a simple main function using Python that sets up a listener on port 1080 on local host:

```
def main():
    modem = QuectelModem(SERIAL_PORT, BAUD_RATE)
    srv   = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.bind(("0.0.0.0", 1080))
    srv.listen(5)
    logging.info("CatSocks proxy listening on 0.0.0.0:1080")

    while True:
        client, addr = srv.accept()
        logging.info(f"Client connected from {addr}")
        threading.Thread(
            target=handle_client,
            args=(client, modem),
            daemon=True
        ).start()
```

The core SOCKS5 function is handled by handle_client as shown below, which implements a minimal SOCKS5 server that supports unauthenticated SOCKS5 CONNECT for both IPv4 and domain names. For each client connection, it parses the SOCKS5 CONNECT request to obtain the target addr:port

```
def handle_client(client_sock, modem):
    try:
        # --- SOCKS5 handshake ---
        ver, nmethods = struct.unpack("!BB", client_sock.recv(2))
        client_sock.recv(nmethods)
        client_sock.sendall(b"\x05\x00")

        # --- CONNECT request ---
        hdr = client_sock.recv(4)
        _, cmd, _, atyp = struct.unpack("!BBBB", hdr)
        if cmd != 1:
            return
        if atyp == 1:       # IPv4
            addr = socket.inet_ntoa(client_sock.recv(4))
        elif atyp == 3:     # Domain
            alen = client_sock.recv(1)[0]
            addr = client_sock.recv(alen).decode()
        else:
            return
        port = struct.unpack("!H", client_sock.recv(2))[0]
        logging.info(f"CONNECT {addr}:{port}")
```

Once the SOCKS5 request is parsed, the following block opens the upstream TCP socket via AT and completes the SOCKS5 response. From the client's point of view, this behaves like a normal SOCKS5 proxy. The fact that the upstream path is a cellular module over UART is fully hidden.

```
# --- Open TCP socket ---

if not modem.open_tcp_direct_push(addr, port):

    client_sock.sendall(b"\x05\x01\x00\x01" + b"\x00"*6)

    return

client_sock.sendall(b"\x05\x00\x00\x01" + b"\x00"*6)
```

The final part is the relay loop, which ties it all together, opening a corresponding TCP connection on the Quectel cellular module using AT+QIOPEN in direct-push mode. This relays data bidirectionally between the client and the modem using AT+QISEND for uplink and +QIURC: "recv" URCs for downlink. It is fully transparent at the TCP level with no TLS termination or inspection. HTTPS/SSH/etc. are just obscure bytes. The net effect is a transparent TCP tunnel over the cellular module, driven by +QIOPEN, +QISEND, +QICLOSE, and +QIURC notifications.

```
client_fd  = client_sock.fileno()
serial_fd  = modem.ser.fileno()
client_sock.setblocking(False)

while True:
    rlist, _, _ = select.select([client_fd, serial_fd], [], [])

    # a) Client → Modem
    if client_fd in rlist:
        data = client_sock.recv(4096)
        if not data:
            break
        # send in MAX_CHUNK_SIZE slices
        offset = 0
        while offset < len(data):
            chunk = data[offset:offset+MAX_CHUNK_SIZE]
            offset += len(chunk)
            logging.info(f"Client→Modem: {len(chunk)} bytes")
            modem.send_raw(chunk)

    # b) Modem → Client
    if serial_fd in rlist:
        # read URC header line
        line = modem.ser.readline().decode(errors='ignore').strip()
        if line.startswith('+QIURC') and 'recv' in line:
            # parse length and read exactly that many bytes
            length = int(line.split(',')[-1])
            payload = b''
            while len(payload) < length:
                payload += modem.ser.read(length - len(payload))
            logging.info(f"Modem→Client: {len(payload)} bytes")
            client_sock.sendall(payload)
        elif 'closed' in line:
            logging.info("Remote closed")
            break

modem.close_tcp()
```

The limitation and characteristic of this initial PoC is a single Quectel socket ID (sock_id=0), which is effectively one real upstream TCP tunnel at a time with no authentication, no UDP, and no BIND support. With one thread per client, sharing one modem, concurrency is limited by the modem's single socket in this design. In the case of the Quectel cellular module used during this research, a total of 12 sockets are available for use, and a multi-socket PoC code supporting multiple sockets has been developed and is available on GitHub.

Metasploit proxy:

The success of the SOCKS5 proxy written in Python motivated us to work on developing a SOCKS5 proxy for Metasploit. By creating a SOCKS5 proxy module, this will allow Metasploit the ability to pivot attacks directly from Metasploit through a physical UART connection on the host system running Metasploit.

In this first example, we focused on targeting devices which use Quectel cellular modules with the plan to add more cellular-based device support over time.

```
msf auxiliary(server/quectel_socks5) >
[*] [15:48:18.732] Configuring serial port /dev/ttyUSB0 to 115200 via stty
[*] [15:48:18.737] Probing modem with AT until OK…
[*] [15:48:35.278] [URC] RDY
[+] [15:48:37.966] Startup AT probe successful (OK).
[*] [15:48:37.966] Waiting briefly for RDY URC (best-effort)…
[*] [15:48:38.187]  Quectel SOCKS5 proxy listening on 0.0.0.0:1080 (sids=12, chunk=1024B)
[*] [15:48:38.188] Started service listener on 0.0.0.0:1080
[*] [15:48:38.188]  listener started, waiting for SOCKS5 clients (Ctrl-C or jobs -k to stop)…
```

In general, the complexity of creating a working Metasploit SOCKS5 proxy for cellular interaction over UART was much more difficult than we initially expected. But in the end, we were successful in creating a working PoC SOCKS5 module to communicate AT socket commands over serial to a Quectel cellular module.

The most complex area was handling overall timing between all the components. This included various interactions and the flow within Metasploit to establish SOCKS5 communications, convert an incoming socks connection into cellular socket commands, and manage this communication over serial UART connection. All while properly sequencing this and dealing with the nuances of a cellular connection in which we could lose connections at time or, in this case, an embedded device ecosystem under test where the cellular module is managed and powered up and down by the CPU. To help maintain this flow within Metasploit we created several timing and control features which we discuss below in more detail.

Quectel SOCKS5 proxy runtime parameters: behavior, limits, and tuning guidance

This section documents the runtime configuration parameters exposed by the Quectel SOCKS5 Metasploit module. These parameters control concurrency, buffering, timing, and flow control between the SOCKS5 client, the host system, and the Quectel cellular modem's AT socket interface within Metasploit's msfconsole.

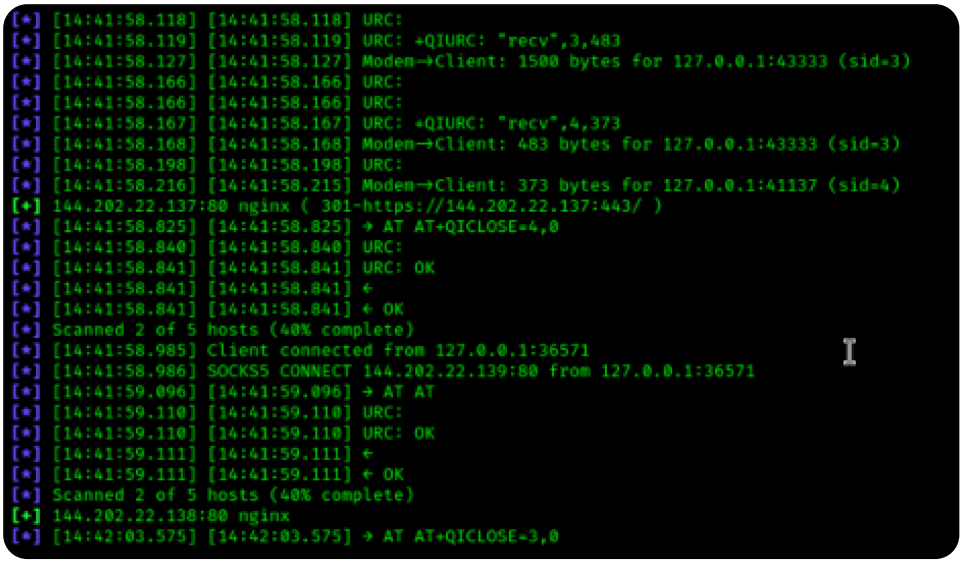
Because Quectel AT socket implementations are not equivalent to a full TCP/IP stack, careful tuning of these parameters is required to achieve stable multi-socket performance without triggering modem-side errors, stalls, or silent drops. Since Quectel AT socket operation is managed by strict firmware-level constraints that are not always clearly documented, stable operation depends on conservative buffer sizing, explicit timeouts, and careful scheduling of send-and-receive operations.

To accomplish this, and provide finer control over these parameters, we made most of these options configurable via msfconsole using the options command. Below is a list of the options we created for the Metasploit module to add more managed control over its timing and flow. The default values currently set, however, should be sufficient to allow normal operations.

- **MODEM_BACKOFF_S (default: 30 s)**
  When the modem is NOT READY, this defines how long a new SOCKS CONNECT request may be held (delayed) while waiting for the modem to recover. Set to 0 to disable holding and fail fast instead.
- **MODEM_BACKOFF_POLL_S (default: 1 s)**
  Polling interval should be used while a SOCKS CONNECT request is being held during modem backoff. Lower values reduce recovery latency but increase looping/serial activity.
- **MODEM_SOCKETS (default: 12 sockets)**
  This defines the maximum number of simultaneous AT socket IDs managed by the module. While Quectel modules typically advertise support for up to 12 sockets, internal scheduling often results in only 2–4 sockets actively transferring data at any given time. This behavior is firmware-dependent and not an error condition.
- **MAX_CHUNK_SIZE (default: 1024 bytes)**
  This specifies the maximum payload size sent per AT+QISEND command. Values above 1024 bytes frequently result in missing '>' prompts or immediate ERROR responses from the modem. This limit is enforced internally by many Quectel firmware builds, making 1024 bytes the most reliable cross-platform setting. This was created to be configurable, to allow future expansion to other cellular module brands which may support a large payload size.
- **SELECT_TIMEOUT_MS (default: 20 ms)**
  This controls the polling interval used when monitoring client sockets for readable data. Lower values improve responsiveness and throughput but increase CPU utilization. Higher values reduce CPU load at the cost of increased latency and reduced parallel efficiency.
- **PROMPT_TIMEOUT_MS (default: 5000 ms)**
  This defines how long the code will wait for the modem's '>' prompt after issuing AT+QISEND. This protects against deadlocks when the modem silently rejects a send operation due to internal buffer pressure or socket state issues.
- **ACK_TIMEOUT_MS (default: 8000 ms)**
  This is the maximum time to wait for SEND OK or SEND FAIL following payload transmission. Cellular network latency, carrier throttling, and radio retransmissions can all delay acknowledgments even after data is accepted by the modem.
- **OPEN_TIMEOUT_MS (default: 30000 ms)**
  This is the maximum wait time for the +QIOPEN URC indicating socket establishment. This includes DNS resolution, PDP context validation, and carrier routing delays. Aggressively lowering this value will cause false failures on congested networks.

- **CMD_TIMEOUT_MS (default: 6000 ms)**
  Timeout should be applied to general AT commands such as ATE0 and QICLOSE. Persistent timeouts typically indicate modem instability, serial contention, or firmware lockups.
- **RECV_DRAIN_MAX_CHUNKS (default: 64)**
  This limits how many receive chunks are drained per socket during a single relay loop iteration. This prevents a single high-bandwidth socket from starving other active sockets, and ensures fair scheduling across connections.
- **STARTUP_OK_TIMEOUT_S (default: 0**
  Startup probe: total seconds to wait for the first successful AT response (OK). A value of 0 disables the timeout (wait indefinitely), which is useful in lab environments where the modem may take longer to boot or recover.
- **STARTUP_OK_INTERVAL_MS (default: 1000 ms)**
  Startup probe: delay between consecutive AT probes while waiting for the first OK. Lower values detect readiness sooner but increase serial traffic.
- **HEALTHCHECK_INTERVAL_S (default: 3 s)**
  Runtime watchdog: frequency of periodic AT probes used to confirm the modem remains responsive during normal operation.
- **HEALTHCHECK_TIMEOUT_MS (default: 2000 ms)**
  Runtime watchdog: per-probe timeout for the modem health-check AT command. Repeated timeouts often indicate modem instability, serial contention, or a crash/hang.
- **HEALTHCHECK_MAX_FAILS (default: 3)**
  Runtime watchdog: number of consecutive failed health probes required before the module marks the modem as NOT READY. When NOT READY, the module stops allocating new Socket ID's (SIDs) and can hold new SOCKS CONNECT requests (depending on backoff settings).

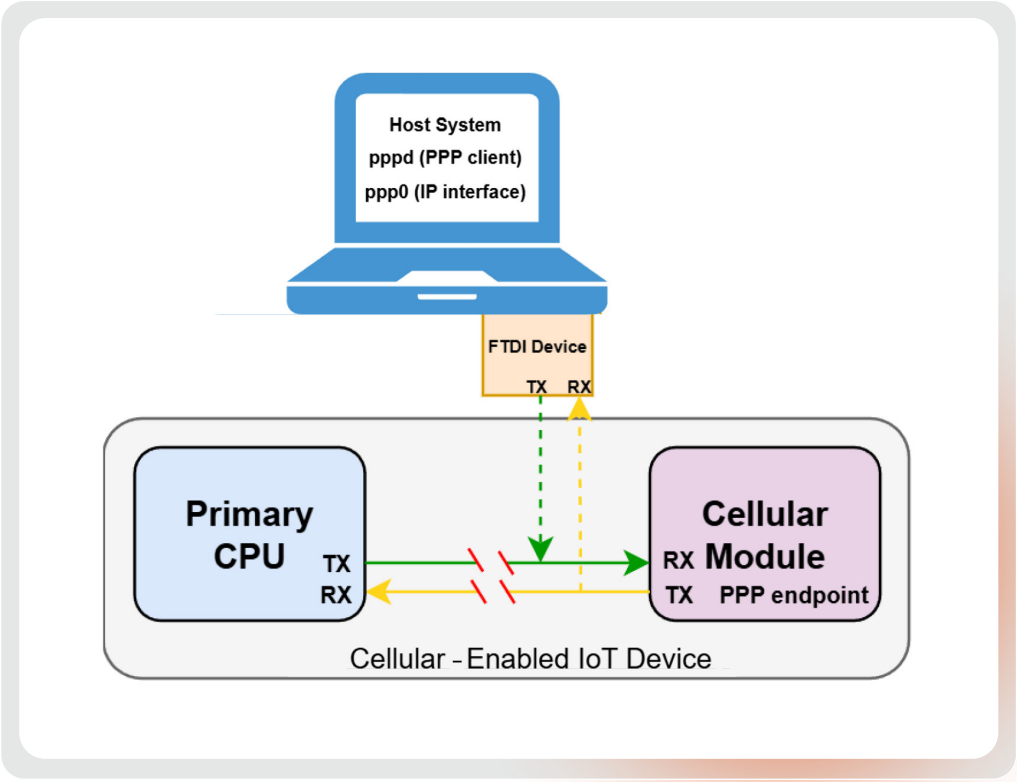
We created an advanced verbose mode, which can be enabled during operation, by running set VERBOSE true for the purpose of detailed monitoring of the SOCKS5 proxy communications between the cellular module and Metasploit modules being run, or any external connected SOCKS5 clients. Enabling verbose monitoring allows for identifying any issues related to timing and communications. Below, is a sample showing verbose enabled for the Cellular_SOCKS5 proxy module while running the Metasploit auxiliary module http_version. This shows all modem checks, along with socks connection and data transfers during the module execution, along with time stamps.



Point-to-Point Protocol (PPP) over UART:

In addition to the SOCKS5 Proxy POC, we explored the cellular module's support for Point-to-Point Protocol (PPP) over UART, to establish a network connection, and route client traffic. To use PPP over UART for network access, the cellular module must be instructed to configure the network context and then placed in data mode. Once in data mode, our host system runs a PPP client, such as the Point-to-Point Protocol daemon (pppd), which initiates a PPP session and establishes an IP interface.

The diagram below provides a high-level overview of the PPP over UART setup, showing the host system running pppd, the cellular module on the device implementing a PPP endpoint, and the FTDI device acting as an intermediary that bridges the host system to the cellular module's PPP endpoint after the original CPU and cellular modem UART connection has been severed.



Before developing our PPP over UART POC, we had to ensure our host had a PPP client. While pppd is available for macOS and Linux, there is no native PPP client support on modern Windows versions. As a result, we used macOS and Linux for development and testing. The pppd System Manager's Manual[23] provides a list of frequently used options, including those for establishing a serial connection, as well as options that allow for finer-grained control of the connection. Additionally, we found tutorials, such as "A 10-Minute Guide for Using PPP to Connect Linux to the Internet,"[24] helpful for determining which options were required in the '/etc/ppp/options' file, which supplies instructions for pppd when setting up a connection. When invoking pppd, several phases occur to establish the connection.

23 https://man.openbsd.org/pppd.several,8
24 https://www.linuxjournal.com/article/210prevent9

They are explained below, along with the options we used within the '/etc/ppp/options' file for configuring the connection:

- **Open Serial Layer:** At this phase, pppd opens the serial transport used for communicating with the cellular modem. This includes specifying the TTY device and baud rate, as well as applying serial-layer options such as 'lock,' which prevents other processes from opening the device, and 'crtscts,' which enables hardware flow control on UART. This phase assumes the cellular mode has already been set to data mode, and no PPP negotiation occurs during this phase.

```
# Open Serial Layer Options

/dev/tty.usbserial-A5069RR4 # serial interface to cellular modem

115200 # baud rate

crtscts # enable hardware (RTS/CTS) flow control

modem # enable modem line handling

lock # prevent other processes from opening TTY
```

- **Link Control Protocol (LCP) Negotiation:**[25] This phase is the first part of the PPP to run. During LCP negotiation, pppd negotiates with the cellular modem the structure of PPP frames on the link, including parameters such as Address-and-Control-Field-Compression, the Async-Control-Character-Map, Protocol-Field-Compression, and related options that define how packets are framed and transmitted over the PPP link.

```
# LCP Negotiation

asyncmap 0 # request no RX control-character escaping

# The below options were set to facilitate traffic analysis in the case
there were issues running tools and did not appear to have a significar
impact on network speeds, but are not required for PPP over UART to
function.

noaccomp # disable address/control field compression

nopcomp # disable protocol field compression

novj # disable Van Jacobson TCP/IP header compression
```

- **PPP Authentication:** Once LCP negotiation is completed, the PPP authentication phase begins. Depending on whether the cellular modem's PPP implementation requires authentication, the host may be required to authenticate. When PPP authentication is required, it is performed using either the Password Authentication Protocol (PAP) or the Challenge Handshake Authentication Protocol (CHAP).

```
# PPP Authentication

user "admin"

password "admin"

# If no authentication is required, use noauth. For the trail camera, I
authentication was required, although any credentials appeared to grant
access.
```

25 https://datatracker.ietf.org/doc/html/rfc1172#page-1

- **IP Control Protocol (IPCP):** In this phase, PPP transitions into an IP network interface by negotiating IP-layer parameters with the cellular modem. Through IPCP, the modem assigns the local and remote IP addresses and provides DNS configuration, which the host accepts to bring up the ppp0 interface and enable IP connectivity.

```
# IPCP

noipdefault # do not assume a local IP address

usepeerdns # accept DNS servers from IPCP

defaultroute # install default route via ppp0
```

In addition to the options listed under their respective phases, we used several others that controlled pppd's runtime behavior and persistence.

```
# Logging and debug

nodetach # Run pppd in the foreground
```


```
# Logging and debug

nodetach # Run pppd in the foreground
```


```
dump # Print the options values to the terminal

# Connection scripts

connect "/usr/local/bin/chat -v -f /pathto/connect.chat"

# Retry behavior

persist # Reopen connection if terminated
```

While pppd can be run from the command line, it requires the cellular module to already be in data mode. As a result, we created a simple Python script to automate some of these steps. The Python script takes two arguments, the serial device and baud rate, to set the cellular module into data mode by confirming it is up and then issuing ATD*99# to set the module into data mode and run pppd.

```
def main():
       parser = argparse.ArgumentParser(description="Start PPP connection via
modem")
    parser.add_argument("--device", required=True, help="Serial device (e.g.
/dev/tty.usbserial-XXXX)")
    parser.add_argument("--baud", type=int, default=115200)
       args = parser.parse_args()
       print(f"Opening {args.device} @ {args.baud}")
       proc = None
       try:
       ser = serial.Serial(args.device, args.baud, timeout=1)
       except serial.SerialException as e:
        restore_default_route()
       sys.exit(f"Could not open serial port: {e}")
       try:
       send_at(ser, "AT")
       send_at(ser, "ATE0")
       resp = send_at(ser, "ATD*99#", delay=3)
       if "CONNECT" not in resp:
            restore_default_route()
              sys.exit("Modem failed to connect: CONNECT not received")
       print("Launching pppd")
       proc = subprocess.Popen(["sudo", "pppd", "connect", "true"])
```

In the final line of the above code block, the Python script launches pppd with the required elevated privileges and the option “connect true.” Normally, pppd uses a connect script, typically implemented via the chat command, that issues AT commands to place the cellular modem in data mode before initiating PPP negotiation. However, we found that relying on a connect script was unreliable due to the sensitive timing of the cellular module and possible noise on the serial line. These issues resulted in stalled connections or failures during LCP negotiation.

To mitigate this, we relied on the Python script to open the serial line to the cellular module, place it into data mode, and then invoke pppd to establish the PPP over UART connection. As a result, pppd is launched with “connect true,” which effectively skips the execution of the connect script and proceeds to PPP negotiation using the parameters in the “/etc/ppp/options” file. This resolved the connectivity issues by separating the steps of placing the cellular module into data mode and establishing the PPP over UART connection.

After pppd is invoked and all PPP protocol phases are completed, the cellular link is exposed to the host as a fully functioning IP network interface. At this point, the host has received an IP address, and the system routing table is updated to direct traffic through the cellular-enabled IoT device, as shown in the image below. This allows standard IP routing, name resolution, and application-layer traffic to be transparently routed through the cellular module.

```
Sun Aug  3 10:35:33 2025 : local  IP address 100.64.181.18
Sun Aug  3 10:35:33 2025 : remote IP address 10.64.64.64
Sun Aug  3 10:35:33 2025 : primary   DNS address 1.1.1.1
Sun Aug  3 10:35:33 2025 : secondary DNS address 8.8.8.8
Sun Aug  3 10:35:33 2025 : Received protocol dictionaries
Sun Aug  3 10:35:33 2025 : Received acsp/dhcp dictionaries
Sun Aug  3 10:35:33 2025 : Committed PPP store
Sun Aug  3 10:35:33 2025 : Received acsp/dhcp dictionaries
Sun Aug  3 10:35:33 2025 : Committed PPP store
Default route is now through ppp0:
default            link#19            UCSg                ppp0
```

Security mitigation

Security mitigation methods to protect cellular-based IoT ecosystems against targeted exploitation can come in several different forms. Unfortunately, some of the methods we discuss in the following paragraphs did not appear to be available on the technology we examined. Nonetheless, cellular module vendors should consider those features as baseline improvements to improve the overall security posture of the technology.

Tamper protection: Tamper protection is a method of securing a hardware device to prevent unauthorized opening of the device. Any attempt to open the device would trigger the tamper protection, rendering the device inoperable. None of the devices we examined during our research, including automotive telemetry units, had any form of tamper protection in place. In general, we would not expect this method to be used on general consumer-grade technology. However, if the technology leverages private APN access, allowing access to backend critical network infrastructure, we would highly recommend that tamper protection technologies be considered to help protect from the physical hardware exploitation-style attacks we discussed within the paper.

Disable unused functions: As discussed in the previous section, cellular modules support both a Main UART and USB for sending commands and data to the cellular module, but typically only one of those is used. As a result, the best solution would be to disable the unused one. Currently, we do not know of a built-in mechanism for doing this; we did not encounter one during our testing of various brands of NB-IoT and LTE-M cellular modules. Here, we would recommend that cellular vendors consider building this feature into current and future cellular module software. Also, we would recommend that during the deployment of this technology, the unused communication path not be connected or made easily accessible in the circuit board design.

Communication encryption: Although we know of no known method that allows encryption of cellular commands from the CPU, data being sent from the CPU through the cellular module can be encrypted and should establish encrypted communication to all the internet, cloud, and backend services and systems. Relying on cellular network encryption is insufficient and can result in data potentially being sent over the internet unencrypted. None of the devices we examined over the last several years encrypted the data leaving the CPU. This made it simple for us to capture data between the CPU and the cellular module to extract passwords, tokens, and keys, which could be replayed directly over the internet to gain access to various services. This is one of the highest areas of risk when ECM is used over USB and it is imperative that the data being transmitted over ECM USB be encrypted before being transmitted. In the case of serial communication through the cellular module, both Main UART and ACM (serial over USB), most cellular modules we tested support encryption through various protocols such as HTTPS and MQTT.

Cellular APN monitoring: When deploying cellular-enabled devices, APN selection (Public and Private) has a direct implication for an organization's ability to monitor and respond to security incidents. Public APNs provide ease of deployment but limit the ability to inspect or control device communications beyond basic usage statistics. In such environments, security monitoring must rely on indirect indicators, such as abnormal data usage, unexpected connections, or deviations from normal and expected communication schedules. These signals, while limited, can still be valuable when integrated into broader security monitoring and should be leveraged when possible.

On the other hand, private APNs enable a more proactive security posture by allowing organizations to apply traditional network monitoring and access control techniques to cellular traffic. This includes enforcing destination constraints, collecting flow-level telemetry, and integrating cellular traffic into existing intrusion detection and logging solutions. For high-risk or high-value deployments, private APNs allow organizations to treat cellular devices as managed network assets rather than obscure endpoints, reducing the likelihood that compromised devices can operate undetected.

Internal network security:[26] [27] [28] When cellular-based IoT technology solutions connect to backend private network segments or private cloud environments, it is imperative that sound network security best practices are deployed. This should include each of the following:

- Network segmentation: This prevents compromised systems from identifying and gaining access to other critical systems or adjacent network segments by enforcing logical and physical separation of resources.
- Network monitoring: Continuous monitoring of network traffic enables detection of anomalous communication patterns, unexpected destinations, and bandwidth usage that may indicate device compromise or misuse.

26 https://www.rapid7.com/fundamentals/what-is-network-security/
27 https://netwrix.com/en/resources/guides/network-security-best-practices/?utm_source=chatgpt.com
28 https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-213.pdf?utm_source=chatgpt.com

- Logging and auditing: Centralized logging and periodic auditing of network activity provide visibility into device communications and support incident investigation, forensic analysis, and compliance requirements.
- Access control enforcement: Strict access controls limit cellular-connected devices to only the network services and resources required for their intended function, reducing the impact of a compromised device.
- Outbound traffic restrictions: Restricting outbound connections to approved IP addresses, domains, and ports helps prevent unauthorized data exfiltration and command-and-control communications.
- Anomaly and behavior detection: Establishing baseline network behavior for devices enables the identification and investigation of deviations such as unusual session duration, traffic volume, or communication timing.
- Product security testing: One of the best methods to identify potential security issues with a given cellular-based IoT product is through security testing. Whether the testing is done by in-house security experts or third-party services,[29] it is critical that all IoT ecosystems are tested on a regular basis or when any new changes or alterations of the product's ecosystem have been made.

While many of these mitigation techniques may not currently be available or widely implemented in existing cellular modules and IoT deployments, they should be considered baseline security requirements for future designs. Controls such as tamper protection, reduction of exposed interfaces, end-to-end encryption, and robust network security practices reduce the attack surface and increase the effort required to successfully exploit the environments that these devices have trusted access to. Organizations leveraging private APN architectures or allowing cellular devices to access internal networks must treat these devices as privileged entry points and apply security controls accordingly.

Ultimately, securing cellular-based IoT technologies requires a holistic approach that acknowledges both physical and logical attack vectors. By combining device-level hardening with strong network monitoring, segmentation, and enforcement, vendors and operators can significantly improve the overall security posture of their deployments. As cellular technologies continue to evolve and proliferate across industries, proactive adoption of these mitigation strategies will be critical to reducing systemic risk and preventing cellular connectivity from becoming an unmonitored and highly trusted attack vector.

29 https://www.rapid7.com/services/iot-security-consulting/

CONCLUSION

While our previous research focused on developing testing methodologies for cellular-enabled IoT devices, we built on that foundation by examining how the access and insights gained from those techniques could enable weaponization. We demonstrated how native AT commands can be used to control a cellular module on an IoT device, extending its functionality beyond its programmed role.

Cellular-enabled IoT architecture relies on internal and external trust relationships between device components and associated backend services. Due to insufficient protections on the cellular-enabled IoT device, we found that the internal trust relationship between the cellular module and primary CPU could be leveraged to gain access to communications and control the cellular module through hardware modification. Using the tools developed during this research, we demonstrated how a cellular-enabled IoT device could be leveraged as an attack vector, creating the potential to abuse existing external trust relationships with cloud services and backend systems.

By examining how cellular-enabled IoT devices function internally and externally, this research highlights how hardware and architectural assumptions directly influence security risk. As cellular-enabled IoT devices continue to proliferate, evaluating these systems from an ecosystem perspective highlights the importance of defining trust boundaries with appropriate security controls during design and integration. This work further demonstrates how access to device hardware can undermine implicit trust assumptions, reinforcing the need to evaluate these boundaries in secure IoT design and deployment.

ABOUT RAPID7

Rapid7, Inc. (NASDAQ: RPD) is a global leader in AI-powered managed cybersecurity operations, trusted to advance organizations' cyber resilience. Open and extensible, the Rapid7 Command Platform integrates security data, enriching it with AI, threat intelligence, and 25 years of expertise and innovation to reduce risk and disrupt attackers. As a recognized leader in preemptive managed detection and response (MDR), Rapid7 unifies exposure and detection to transform the cybersecurity operations of more than 11,500 customers worldwide. For more information, visit our website, check out our blog, or follow us on LinkedIn or X.