Modware: Securing the Modbus Protocol

Mohammad Eshan[1] and Chris Tremblay[2]

{me3031[1],cst1465[2]}@rit.edu

https://github.com/cstremblay2000/Modware

Contents

Modware: Securing the Modbus Protocol

Christopher Tremblay
Department of Computing Security
College of Computing and Information Sciences
Rochester Institute of Technology
cst1465@rit.edu

Mohammad Eshan
Department of Computing Security
College of Computing and Information Sciences
Rochester Institute of Technology
me3031@rit.edu

Abstract

This research paper presents a novel middleware solution designed to bolster the security of the widely-used Modbus protocol in industrial control systems. Despite its pervasive usage, the Modbus protocol is known to lack essential security features, rendering devices and infrastructure vulnerable to cyberattacks. Our proposed middleware solution addresses this issue by incorporating advanced cryptographic standards for encryption, signing, and verification operations, leveraging RSA with 3072-bit keys for public key encryption and signing. This approach eliminates the need for modifying existing hardware, requiring only minimal reconfiguration for implementation. As such, it offers an unobtrusive and efficient method for securing communications. This innovative solution enables enhanced security for Modbus communications without necessitating the adoption of Secure Modbus or TLS, effectively ensuring confidentiality, integrity, non-repudiation, and authentication.

1 Introduction

Industrial Control Systems (ICS) are essential in managing and maintaining the efficient and safe operation of critical infrastructure, including power plants, water treatment facilities, and transportation systems. One widely-used communication protocol within these systems is Modbus, which enables data exchange between devices, such as sensors and actuators, and their controllers. However, Modbus was developed in an era when security was not a primary concern, and as a result, it is inherently vulnerable to a variety of attacks, including unauthorized access, data tampering, and denial of service. These vulnerabilities leave critical infrastructure exposed to potential breaches, with far-reaching consequences for public safety and economic stability.

Recognizing the need for enhanced security, the Modbus Organization has proposed a solution called Secure Modbus, which leverages Transport Layer Security (TLS) versions 1.1 and 1.2 to encrypt data and authenticate devices. However, the adoption of Secure Modbus has been slow among many sensors and industrial equipment manufacturers due to the increased complexity and overhead associated with implementing TLS at a large scale. Additionally, retrofitting existing equipment with Secure Modbus can be cost-prohibitive for many organizations, further hindering its widespread adoption.

To address these challenges, this paper presents a novel middleware solution that seamlessly encrypts Modbus communication without requiring modifications to existing hardware. This approach not only enhances security by adding a robust layer of encryption and authentication but also minimizes the impact on system performance, making it an unobtrusive and efficient alternative to Secure Modbus. Furthermore, our middleware solution is designed to be scalable and adaptable, allowing it to accommodate various industry-specific requirements and remain compatible with future advancements in encryption technology.

By offering a more accessible and cost-effective means of securing Modbus communication, our middleware solution can facilitate the broader adoption of robust security measures across various industries, ultimately contributing to the overall resilience of critical infrastructure in the face of evolving cyber threats. Through extensive experimentation and real-world case studies, we demonstrate the effectiveness of our middleware approach in securing Modbus communication while maintaining high performance and operational efficiency, highlighting its potential to become a key component in the ongoing efforts to safeguard critical infrastructure from cyber-attacks.

2 Background

2.1 Modbus Protocol

The Modbus protocol is a simple and effective method of communication between client and server devices in industrial automation systems. Typically, servers consist of sensing devices, while clients are controllers interested in the values of those sensors. Modbus supports two primary communication modes: a query/response method for direct communication between client and server, and a broadcast communication mode where a client sends a command to all servers [10].

To distinguish between commands sent to a client, Modbus employs function codes. For instance, the function code 0x01 corresponds to the ReadCoils function, which is used to retrieve the status of specified coils in the remote device. In response, the remote device echoes back the same function code, along with ON/OFF values encoded as single bits.

2.2 Modbus and Security

Modbus was first published in 1979, during a time when industrial control systems were isolated, and networking them was impractical. Consequently, the designers of Modbus did not incorporate basic security features such as encryption, authentication, and integrity in the protocol's design [14]. Today, Modbus and similar IoT/SCADA protocols, such as Profibus and DNP3, are vulnerable to attacks due to changes in the threat landscape since their inception. These protocols were created during an era before security was a major concern.

Nonetheless, Modbus was chosen for this project due to its popularity in industrial settings, its open-source nature, and its availability at no cost. While Modbus was not originally designed with security in mind, there are various measures that can be employed to secure Modbus communications. For example, using virtual private networks (VPNs) can encrypt and authenticate communication between client and server devices, while firewalls can restrict access to the Modbus network. Additionally, secure Modbus variants, such as Modbus/TCP Secure and Modbus Secure, offer encryption, authentication, and integrity protection. Regular security assessments and penetration testing can also help identify and address potential vulnerabilities in the Modbus network.

3 Related Work

The issue of Modbus security has received limited attention in academic literature. Various solutions have been proposed, with some recurring themes and methodologies. In this section, we discuss the different approaches, highlighting their advantages and limitations.

3.1 Adding TLS to the Communication Stack

One recurring solution observed in the literature is the incorporation of Transport Layer Security (TLS) into the communication stack. The Modbus Organization themselves provided the protocol specification [5], and researchers from China also proposed a similar solution [26]. Adding TLS to the communication stack is a viable option, as it ensures encryption, authentication, and integrity protection. However, legacy and lower-end equipment may not have a sophisticated enough communication stack to implement this solution, nor the resources to perform both their designed tasks and the computationally-intensive cryptographic operations required by TLS.

3.2 Middleware and Node-RED

The authors in [22] propose a comprehensive solution to Modbus security that supports roles, privilege levels, and non-repudiation of human users. Their approach relies on a server running middleware between the communicating devices, which acts as a man-in-the-middle (MITM) to facilitate all security functions. This solution was built on top of a software package called Node-RED, which already provides a subset of security features. While this is a sophisticated and feature-dense solution, it may be less desirable for companies that do not require the entire security suite and view the additional features as unnecessary bloat.

3.3 Modified Modbus Protocol with MITM Server

The authors in [16] propose a system that slightly modifies the Modbus protocol and employs a MITM server. They suggest appending a digest to the end of the Modbus packet, signing the entire packet with an RSA private key, and including a timestamp within the digest. These features provide integrity, non-repudiation, and freshness against replay attacks. However, similar to the approach in [4], this solution requires a server running middleware to facilitate communication between end devices.

In summary, the existing literature presents various approaches to enhancing Modbus security, including adding TLS to the communication stack, employing middleware solutions, and modifying the Modbus protocol itself. However, each of these solutions has its limitations, particularly with respect to compatibility with legacy and lower-end equipment.

3.4 Intrusion Detection Systems

Intrusion Detection Systems (IDS) have been proposed as a method to identify and mitigate cyber threats targeting Modbus communication [13]. These systems can be either signature-based or anomaly-based, and they monitor network traffic for signs of malicious activity. While IDS solutions can be effective in detecting and preventing unauthorized access, they do not inherently provide encryption, authentication, or integrity protection for Modbus communication. Furthermore, IDS solutions may introduce latency, require continuous updates, and suffer from false positives and negatives.

3.5 Application of Blockchain Technology

Blockchain technology has been suggested as a potential solution for securing Modbus communication [1]. By leveraging distributed ledger technology, blockchain can enable secure and tamper-proof record-keeping of Modbus transactions, as well as provide cryptographic mechanisms for device authentication. However, the application of blockchain in this context can introduce significant overhead, both in terms of computational and communication resources, which may be prohibitive for legacy and resource-constrained systems.

3.6 Physical Security Measures

Physical security measures, such as secure hardware modules and tamper-resistant enclosures, can complement cybersecurity efforts to protect Modbus communication [18]. These measures can prevent unauthorized physical access to devices and ensure the secure storage of cryptographic keys. Although physical security measures can provide an additional layer of protection, they may be insufficient to fully secure Modbus communication, particularly against remote cyber threats.

In conclusion, the existing literature presents a diverse array of approaches to enhancing Modbus security, ranging from the application of TLS and middleware solutions to intrusion detection systems, blockchain technology, and physical security measures. While each solution offers unique benefits, none provides a comprehensive and universally applicable answer to the security challenges posed by the Modbus protocol. As industrial control systems continue to evolve and face increasingly sophisticated cyber threats, further research and innovation will be required to address the complex security landscape and ensure the safe and reliable operation of critical infrastructure.

4 Project Idea

We provide an unobtrusive client/server middleware solution that enables secure communication without intrusive modifications to the existing hardware or extensive time investment for integration. The basic model for the system can be seen in Figure 1. Recognizing that the data transmitted by Modbus devices is intrinsically insecure, our middleware is strategically designed to be positioned in close proximity to the devices, preferably on the same switch or even in line. This approach minimizes the duration of unencrypted traffic, significantly reducing the window of vulnerability. The protocol sequence diagrams outlining the functionality of Modware can be seen in Figures 2 and 3 in Appendix A.

A core objective of our project is to address the security challenges inherent in legacy and resource-constrained industrial systems, which often lack the capacity to support advanced cryptographic processes. By implementing this middleware solution, we strive to offer a seamless yet robust layer of protection for these systems without disrupting their normal operations.

Furthermore, our project aims to facilitate the adoption of secure communication practices in various industrial settings by offering a cost-effective and easily deployable solution. This middleware can be scaled across different environments, making it a versatile security enhancement for a wide range of industrial control systems that rely on protocols with limited built-in security measures.

Ultimately, our research endeavors to contribute to the broader understanding of cybersecurity in industrial contexts, while providing practical solutions for organizations seeking to bolster their security infrastructure without significant disruption or financial burden.

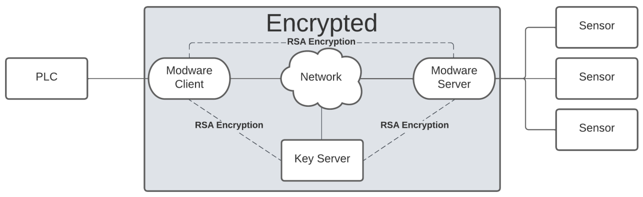


Figure 1: Modware Model

5 Project Implementation

This project was implemented using the `Go` programming language, primarily due to its efficient and versatile standard libraries, which provide robust support for networking, cryptography, error checking, and straightforward byte-level manipulation. We specifically chose `Go` for these advantages. Although `C/C++` is known for its performance and control, we opted against it due to its higher development overhead and complexity. Additionally, while `Python` is popular for its ease of use and rapid development capabilities, it can be less suited for byte-level networking operations.

`Go` offers an ideal balance, combining the networking and byte manipulation strengths of `C/C++` with the user-friendly nature and swift development time of `Python`. This combination enables us to effectively tackle the challenges associated with implementing secure communication middleware for industrial control systems.

Furthermore, our project plans to leverage `Terraform` for the construction of our test environment on the `Microsoft Azure Platform`. Terraform is an Infrastructure-as-Code (IaC) tool that enables the efficient and consistent deployment of infrastructure resources across multiple platforms. By utilizing Terraform, we can streamline the provisioning and management of our test environment, ensuring reproducibility and adaptability to various scenarios.

For the deployment of our project components across different environments, we will employ `Ansible`, a widely-used open-source automation tool. Ansible simplifies the process of deploying, configuring, and managing our middleware solution on various target systems, thereby accelerating the implementation and integration of our project into a diverse range of industrial settings. This approach not only enhances the versatility of our solution but also contributes to its ease of adoption by organizations seeking to improve their security posture without significant disruption or investment.

5.1 Project Architecture

The project is divided into three Go modules: `ModwareClient`, `ModwareServer`, and `KeyServer`. Each module contains several supporting files. For `ModwareClient`, there are two source code files: one that includes all the client-side logic, and one that includes all the functions and structures related to the protocol. Additionally, `ModwareClient` has its public/private keys and the `KeyServer`'s public key in the root directory. Furthermore, there is a `keys/` directory in which the module stores its learned keys. `ModwareServer` has an identical structure, except that the protocol source code file contains only the functionality required by the server. Lastly, `KeyServer` has a similar file structure but includes a `keys/` directory that holds all the public and private keys in the system instance.

5.2 ModwareClient

The ModwareClient is comprised of 3 parts: *initialization*, *Verify Host*, and *Verified Communication*.

Initialization: The ModwareClient loads its stored public and private keys into memory. It binds to a server socket on port 5020, this was done intentionally to avoid conflict with legitimate Modbus traffic. The server socket listens and waits for a real Modbus client device, such as a PLC or operator interface, to make a request.

Verify Host: Upon reception of a request, the ModwareClient checks the destination address of the packet to see if it has the public key for that device. If it doesn't, it begins the client-side portion of the *Verify Host* protocol in Figure 2. First, the ModwareClient sends a request to the ModwareServer with a payload of "MAC" and waits for a response with its MAC address. This is how the KeyServer will help uniquely identify the ModwareServer in case there are IP conflicts. After reception of the MAC address, the ModwareClient opens a socket to the KeyServer. It packages the IP and MAC addresses into a struct, and the gob encodes it into bytes to be sent out over the network. The gob encoding is native to `Go`, therefore all participating middleware within the system must also be implemented in `Go`. If the KeyServer deems that the ModwareClient is allowed to talk to the requested ModwareServer, then it will send a packet back containing a few different things. First, it will contain a unique challenge generated by the KeyServer. Second, it will contain the expected signature of the ModwareServer. Since the KeyServer knows about all keys, it can generate this expected signature. Third, the KeyServer will sign the expected signature so the ModwareClient can verify the work was done by the legitimate KeyServer. Finally, the public key of the ModwareServer is sent along as well for signature validation. The only part that is encrypted from the KeyServer is the challenge due to the size limitations of messages for public key encryption.

Once that four-element structure is received from the KeyServer, the socket is closed to the KeyServer and the ModwareClient waits for reception of the challenge signature from the ModwareServer. Once all packets have been received, the ModwareClient loads its copy of the KeyServer's public key into memory and verifies that the expected signature was signed by the legitimate KeyServer (`Verify( Pks, Sks, sigChall )` in Figure 2). If it isn't valid, then all communication to the KeyServer and ModwareServer is closed immediately. Upon verification of the signature, the ModwareClient then verifies the signature received from the ModwareServer with the public key sent by the ModwareServer. If that is valid, then it performs one last check that the expected and received signatures are valid. If the ModwareServer signature is not verified or the expected and received signatures do not match, then communication is terminated. Upon success, the public key is stored on disk for later use.

In other words, the ModwareClient verifies the expected challenge signature by verifying the signed signature from the KeyServer, thus ensuring that the expected challenge signature can be trusted. Once the ModwareClient has verified the <u>expected</u> challenge signature from the KeyServer, it proceeds to verify the <u>received</u> signature from the ModwareServer with the public key provided by the KeyServer. Then, it compares the two signatures to ensure that they match. Since the expected challenge signature came from a trusted and verified source, the received challenge signature was validated, and the expected and received signatures match, it can be concluded that the ModwareServer is trusted.

Verified Communication: Now that a ModwareServer device has been verified, the ModwareClient device can securely communicate with the protocol depicted in Figure 3. A unique challenge is generated, encrypted with the ModwareServer public key, and sent to the ModwareServer. The received response from the ModwareServer is expected to be the signature of the challenge signed by the ModwareServer. The signature is validated to ensure that it came from the ModwareServer. Upon successful attestation of the challenge, the ModwareClient forwards the Modbus request along. First, an HMAC of the request with the HMAC-SHA-256 algorithm keyed by the challenge is generated. The HMAC and the packet are then packed into a struct, gob encoded into bytes, encrypted, and sent out to the network. Upon receipt of the response, the ModwareClient will decrypt it, decode the gob-encoded struct and pull out the encapsulated Modbus response and HMAC. The Modbus response is verified by generating an HMAC of the response keyed by the challenge and comparing that HMAC to the received HMAC. If they match, then the Modbus response is forwarded to the Modbus client device.

5.3 ModwareServer

The ModwareServer, similar to the ModwareClient, is structured into three components: *initialization*, *Verify Host*, and *Verified Communication*.

Initialization: The ModwareServer, much like the ModwareClient, begins by loading its public and private keys into memory. It establishes a server socket on port 5020 to prevent interference with legitimate Modbus traffic and listens for incoming connections from ModwareClients.

Verify Host: Upon receiving a message, the ModwareServer first examines the payload to determine if it is a MAC address request by converting it into a string. The request is signified by the ASCII string "MAC". If a MAC address request is detected, the ModwareServer adheres to the server-side protocol flow illustrated in Figure 2. Initially, the ModwareServer responds with its MAC address, enabling remote devices to participate in the protocol, even if they are situated far away or on the same local area network.

Subsequently, the ModwareServer binds to a new server socket, awaiting a connection from the KeyServer. Upon receiving the KeyServer's message, the ModwareServer decodes it using gob and converts it into a struct. It then decrypts the encapsulated challenge, signs it, and sends the resulting signature over the network. Finally, the ModwareServer stores the encapsulated public key on disk for future use.

Verified Communication: The ModwareClient maintains a list of known ModwareServers. If the initial message received by the ModwareServer is not a MAC address request, it is assumed that verified communication is taking place and that the ModwareClient has already completed the *Verify Host* process. The ModwareServer then follows the server-side protocol depicted in Figure 3.

The ModwareServer first extracts the source IP address from the packet and uses it to locate the associated public key for the ModwareClient device. It then decrypts the packet, signs the challenge, and sends it out over the network. If the ModwareClient does not terminate the communication and responds, the ModwareServer decrypts the subsequent packet, decodes it using gob into a struct, and extracts the encapsulated HMAC and Modbus request. The ModwareServer verifies the HMAC by generating its own HMAC-SHA-256 digest, using the challenge as the key. If the generated HMAC matches the expected HMAC, the ModwareClient forwards the Modbus request to the device, receives the response, and sends it back to the ModwareServer.

Finally, the ModwareServer generates an HMAC for the Modbus response using the HMAC-SHA-256 algorithm, with the challenge serving as the key. The HMAC and Modbus response are combined into a struct, encoded using gob into bytes, encrypted, and transmitted back to the ModwareClient.

5.4 KeyServer

The KeyServer plays a crucial role in the Modware system by managing and distributing keys. Its functionality is divided into two main components: *initialization* and *Verify Host*. Unlike the ModwareClient and ModwareServer, the KeyServer does not participate in Modbus communication, focusing solely on facilitating secure key distribution.

Initialization: The KeyServer commences its operation by loading its public and private keys into memory. It then establishes a server socket listening on port 5020, which is designated for communication with ModwareClients and ModwareServers.

Verify Host: Upon receiving a message, the KeyServer assumes that a ModwareClient is attempting to verify a ModwareServer with which it is not yet familiar. The KeyServer first decodes the received message using gob decoding and extracts the IP and MAC address from the payload. Utilizing the source IP address of the incoming connection, the KeyServer retrieves the public key of the ModwareClient. Subsequently, the KeyServer leverages the information within the payload to load the public and private keys of the ModwareServer into memory. To facilitate the verification process, the KeyServer generates a unique challenge and signs it using the private key of the ModwareServer. This signed challenge is then sent to the ModwareClient, allowing it to verify the authenticity of the ModwareServer and establish a secure communication channel.

6 Challenges

During the development of this project, several challenges were encountered, predominantly related to the implementation of public key cryptosystems. This section discusses these challenges and the solutions that were employed to address them.

Message Length Limitation: A challenge emerged when attempting to encrypt messages that exceeded the length of the public key. This issue was particularly evident in the *Verify Host* flow, as illustrated in Figure 2. The initial approach involved encrypting the public key, challenge, and both signatures; however, this significantly surpassed the maximum message size for the 3072-bit public keys. In response, we selectively encrypted only the necessary components. While ideally, the entire packet would have been encrypted, in practice, only the challenge required secrecy, and the signatures did not strictly demand encryption. Consequently, we chose to encrypt solely the challenge.

Signature Mismatch: Another challenge arose in the *Verify Host* flow depicted in Figure 2. Within the codebase, the `crypto/rsa.SignPSS()` function is utilized to sign the message, which represents the `Go` implementation of the RSA Probabilistic Signature Scheme. When the KeyServer generates the expected signature using its copy of the ModwareServer's private key, the outcome differs from the received signature created by the actual ModwareServer, even though both are technically valid signatures for the same message. Given that the protocol relies on both signatures being valid and identical, this discrepancy posed a problem. Although the `Go` language recommends using the native cryptographic random number generator (RNG) for the `SignPSS` function to ensure maximum security, we adopted a different approach to maintain secrecy while ensuring consistent pseudorandom number generation. By seeding an RNG with the challenge generated by the KeyServer, we were able to achieve the desired outcome of valid and identical signatures for the message.

7 Experimental Results

Experiments in a simulated cloud environment using `Terraform`, and a physical lab environment were conducted to validate Modware's functionality.

7.1 Physical Lab Environment Results

During the physical lab testing, a Raspberry Pi and three Windows PCs were utilized for the Modware instance. Table 1 provides detailed information on each device and their specific role within the instance, and Figure 4 showcases the lab setup. To simulate the Modbus client device, we utilized Ignition [8], while OpenPLC [2] was used to simulate the Modbus server device. All Figures for the results in this section can be seen in Appendix B.

The first PC, with IP address `192.168.1.2`, was a regular Windows machine with `Go` installed. The ModwareClient software was running on this device. The second PC, with IP address `192.168.1.3`, was both the ModwareClient and Modbus client device packaged into one PC. Both `Go` and Ignition [8] were installed on this PC. The ModwareClient was run on this device, and Ignition was configured to have the local ModwareClient process as a Modbus client device. The third PC, with IP address `192.168.1.4`, was the KeyServer. It was only configured with `Go`. Finally, the Raspberry Pi was configured with OpenPLC [2]. It was also configured with a static route using the `ip` utility to send all of its traffic to the `192.168.1.2` Windows PC.

First, normal Modbus traffic can be observed in Figure 5. This represents standard communication between the OpenPLC device and Ignition. The standard format of Modbus can be seen in this figure. The Modbus ReadCoil Query is highlighted in green, and the associated Modbus Response can be seen in red.

Now to show the security enhancement and functionality of Modware, Figure 6 overviews the protocol flow including *VerifyHost*, and *VerifiedCommunication* (captured from the ModwareServer side). Starting with *VerifyHost* detailed in the sequence diagram in Figure 2, the first red box captures the TCP 3-way handshake from the ModwareClient to the ModwareServer. The following yellow box then details the MAC address request and response. Figures 7 and 8 show the packet contents of the request and response, respectively, more thoroughly. The second red box shows the 3-way handshake between the ModwareServer and KeyServer. The following cyan box shows the KeyServer sending the encrypted challenge to the ModwareServer. Figures 9, 10, and 11 show the packet contents of the ModwareClient public key request, KeyServer public key response to the ModwareClient, and KeyServer public key response to the ModwareServer, respectively. Finally, the blue box demonstrates the ModwareServer sending the challenge signature back to the ModwareClient, and since the signatures were matched and verified the ModwareClient begins the *VerifiedCommunication* flow.

The *VerifiedCommunication* stage begins, starting at the pink in Figure 6. This protocol flow is shown in the sequence diagram in Figure 3. The first packets, in the pink box, detail the ModwareClient to ModwareServer challenge attestation. Packet 268 is the challenge, and packet 269 is the response. Since they are encrypted, there is no need to show the contents of the packet. Following the pink box is the green box, which then details the remainder of the *VerifiedCommunicaton* flow. The ModwareServer forwards the packet to the actual Modbus server device and then takes the response, encrypts it, and sends it back to the ModwareClient. From there the socket is closed until the next Modbus request is sent by the actual Modbus client device. Since the *VerifyHost* flow has already been completed for that ModwareServer, then it won't have to execute again.

7.2 Simulated Results

In the simulated lab environment, we leveraged Microsoft Azure and Terraform to create a virtualized infrastructure for testing the Modware instance. This setup included a virtual Raspberry Pi, three virtual Windows PCs, and a KeyServer. Each virtual device had a specific role within the instance.

Azure Virtual Network (VNet) was utilized to connect all virtual devices, with each device having a unique IP address within the VNet. Terraform scripts were used to automate the creation and configuration of the virtual devices and the network itself.

The first virtual Windows PC, with Go installed, was used to run the ModwareClient software. The second virtual Windows PC was both the ModwareClient and Modbus client device. Both Go and Ignition were installed on this PC, and

Ignition was configured to have the local ModwareClient process as a Modbus client device. The third virtual Windows PC acted as the KeyServer and was only configured with Go. Finally, the virtual Raspberry Pi was configured with OpenPLC.

Normal Modbus traffic was observed in the simulated environment, representing standard communication between the virtual OpenPLC device and Ignition.

In the simulated environment, the protocol flow, including VerifyHost and VerifiedCommunication, demonstrated the security enhancement and functionality of Modware. This part of the simulation closely follows the protocol flow detailed in the Physical Lab Environment Results section.

The simulated environment results confirm that the Modware implementation effectively secures Modbus communication in a virtual environment, as it does in the physical environment. Utilizing Azure and Terraform for virtual infrastructure provisioning and configuration proved to be a viable and efficient approach for testing the Modware instance in a simulated setting.

8 Future Work

The current project provides a strong starting point for securing the Modbus protocol; however, there is room for further enhancements and improvements:

Key Storage Improvement: As previously mentioned, one potential area for improvement is the transition from file-based key storage to a more robust solution. Upgrading to a database for key storage would be a natural progression, offering improved organization, scalability, and flexibility as the number of devices in the system increases. Alternatively, secure cloud storage solutions for key management could be considered. Cloud storage offers several advantages, such as enhanced accessibility, allowing authorized devices to access keys from any location with internet connectivity. Furthermore, cloud storage providers typically incorporate built-in security features, such as encryption-at-rest and in-transit, access control mechanisms, and monitoring capabilities [12]. The use of cloud storage may also simplify key backup and recovery processes, as well as facilitate the implementation of automated key rotation mechanisms. It is essential to weigh the potential benefits of cloud storage against the risks and challenges, such as data privacy concerns, dependency on third-party providers, and potential latency issues, before opting for a cloud-based solution.

Secure Private Key Storage and Handling: Enhancing the security of private key storage and handling is a crucial aspect of future work. As previously mentioned, incorporating a Trusted Platform Module (TPM) or a software-based TPM for secure key management could offer a more robust solution compared to the existing approach. In addition to these options, other cryptographic libraries and protocols, such as the Key Management Interoperability Protocol (KMIP) [17], could be explored for secure key storage and management. KMIP is a universal protocol that standardizes communication between key management systems and cryptographic clients. By adopting KMIP or similar protocols, the system can leverage a standardized, extensible framework for key lifecycle management, ensuring that keys are securely generated, stored, and retired.

Moreover, the use of Hardware Security Modules (HSMs) [3] can be considered for safeguarding cryptographic keys and offloading cryptographic operations from the middleware. HSMs are dedicated, tamper-resistant hardware devices that provide a high level of security for cryptographic operations and key management. By incorporating HSMs into the system, private keys can be securely stored and isolated from the main application, minimizing the risk of key exposure and improving overall system security.

It is important to evaluate the feasibility, cost, and performance implications of implementing these various secure key storage and handling solutions to ensure they align with the specific requirements of the system and the industrial environment in which they will be deployed.

Scalability and Performance Optimization: To ensure the middleware's scalability and performance in larger industrial settings, various optimization techniques and strategies can be explored. One such approach is implementing load balancing [21], which involves distributing workloads across multiple processing resources to prevent any single resource from becoming a bottleneck. Load balancing can be achieved using a combination of hardware and software solutions, helping to optimize resource utilization, minimize response times, and improve the overall system's reliability and fault tolerance.

Another technique that could be employed is parallel processing [11], which refers to the simultaneous execution of multiple tasks or processes on multiple processing elements. By leveraging parallel processing techniques, the middleware can handle increased workloads more efficiently and reduce the time taken to process and secure communications between devices. This can be particularly beneficial in time-sensitive industrial environments, where even small delays in communication can have significant consequences.

Furthermore, optimizing data structures and algorithms within the middleware can lead to performance improvements, ensuring that the system operates efficiently as the number of devices and communication channels grows. Techniques such as caching, memoization, and just-in-time compilation [19] can be employed to speed up frequently performed operations and reduce computational overheads.

Ultimately, a comprehensive approach to scalability and performance optimization will involve evaluating and addressing potential bottlenecks and inefficiencies throughout the middleware and the broader system, ensuring that the solution remains effective and responsive in the face of growing workloads and evolving industrial requirements.

Integration with Other Industrial Protocols: The current project focuses on securing the Modbus protocol, but numerous other industrial communication protocols are prevalent in various industrial sectors. By extending the solution to work with other industrial protocols, the middleware could become more versatile and widely applicable in securing industrial control systems. Some of the key protocols that could be targeted for integration include:

- DNP3 (Distributed Network Protocol 3) [6]: DNP3 is a widely used communication protocol in the electric utility industry, particularly for supervisory control and data acquisition (SCADA) systems. Adapting the middleware to support DNP3 would help secure critical infrastructure in the power grid and enhance overall grid resilience.
- EtherNet/IP (Ethernet Industrial Protocol) [23]: EtherNet/IP is a widely adopted industrial protocol that utilizes the standard Ethernet and the Common Industrial Protocol (CIP) to manage industrial automation applications. Integrating EtherNet/IP compatibility into the middleware would make it suitable for a broad range of manufacturing and process control systems that rely on this protocol.
- PROFINET [24]: As an open standard for Industrial Ethernet, PROFINET is designed for data exchange between automation controllers and devices in real-time. Incorporating support for PROFINET in the middleware would extend its applicability to various industrial automation systems, particularly those in the automotive, aerospace, and manufacturing sectors.
- IEC 61850 [9]: This protocol is specifically designed for the communication and interoperability within electrical substations. By integrating IEC 61850 support, the middleware could help secure and protect critical power infrastructure against cyber-attacks and maintain the stability of power systems.

Extending the solution to support these and other industrial protocols would involve conducting thorough research and analysis to understand the intricacies of each protocol, identify specific security vulnerabilities, and develop tailored strategies for securing communications. This approach would not only make the middleware more widely applicable but also contribute to the overall goal of securing industrial control systems across various sectors and applications.

Automated Key Management and Rotation: Enhancing the security of the system through automated key management and rotation mechanisms is crucial to maintain secure communication channels and protect against potential cyber-attacks. This can be achieved by incorporating existing protocols and standards or developing custom solutions tailored to the specific requirements of the system. Some of the key technologies and approaches that could be explored for this purpose include:

- Internet Key Exchange (IKE) [15]: IKE is a protocol used to establish secure communication channels over an insecure network, primarily in the context of IPsec-based Virtual Private Networks (VPNs). By leveraging the IKE protocol, the middleware could automate the process of negotiating cryptographic keys and security associations, ensuring that keys are regularly updated and reducing the likelihood of successful attacks due to compromised keys.
- Automated Certificate Management Environment (ACME) [20]: ACME is a protocol for automating the management of public key infrastructure (PKI) certificates. In the context of the middleware, ACME could be used to streamline the process of issuing, renewing, and revoking certificates, thereby ensuring that keys remain up-to-date and minimizing the risk of key compromise.
- Custom Key Management and Rotation Solutions: Depending on the unique requirements of the system and the specific security threats it faces, it may be beneficial to develop a custom key management and rotation solution. Such a solution could be tailored to the specific needs of the system, taking into account factors such as the frequency of key updates, the desired level of security, and the potential impact of key compromise on system functionality and integrity.
- Integration with Hardware Security Modules (HSMs) [7]: HSMs are dedicated devices designed to securely generate, store, and manage cryptographic keys. By integrating HSMs into the key management process, the middleware could enhance the security of key storage and handling while also facilitating automated key rotation and management.
- Centralized Key Management Systems [25]: In large-scale industrial environments, centralized key management systems can offer a more streamlined approach to key management and rotation. Such systems could be used to automate key updates, manage access control policies, and monitor key usage, ensuring that keys remain secure and up-to-date throughout their lifecycle.

By implementing automated key management and rotation mechanisms using one or more of these approaches, the middleware could further enhance the security of the system, reduce the risk of key compromise, and maintain the integrity of the communication channels.

9 Conclusion

In conclusion, this study presents an innovative middleware solution aimed at enhancing the security of the Modbus protocol, a prevalent industrial communication protocol lacking inherent security mechanisms. By integrating public key cryptography and digital signatures, the proposed middleware achieves a balance between ensuring secure Modbus communication and maintaining its simplicity and ease of implementation.

Throughout the research and development process, the project faced several challenges, such as message length limitations and signature mismatches. The solutions devised to overcome these obstacles demonstrate the adaptability and effectiveness of the middleware in addressing real-world implementation concerns. The experimental results, involving the deployment of virtual machines on the Microsoft Azure platform, validate the middleware's performance in a simulated industrial setting.

Future work should focus on refining the middleware by enhancing key storage, secure private key handling, scalability, and performance optimization, as well as extending support to other industrial protocols. Furthermore, the exploration and incorporation of automated key management and rotation mechanisms will bolster the security and maintain the integrity of communication channels.

By tackling these improvement areas and adapting the middleware to accommodate a broader range of industrial protocols, the proposed solution can significantly contribute to the security of industrial control systems across diverse sectors and applications. Ultimately, this research and the middleware solution presented herein serve as a crucial step towards safeguarding critical infrastructure, enhancing overall system security, and defending against potential cyber-attacks in the rapidly evolving domain of industrial control systems.

10 Project Contributions

In this project, the contributions were divided between two team members, Mohammad Eshan and Chris Tremblay, with each contributing to various aspects of the project, including development, testing, and documentation. Both team members collaborated effectively to ensure a successful and well-rounded project. Their combined efforts resulted in a comprehensive and informative research paper, as well as a functional and reliable Modware system.

10.1 Mohammad Eshan

Mohammad Eshan primarily focused on the development and the testing utilities for the project. He was responsible for creating the `dev_utils` component, which includes the `mock_server.go` and `modbus_client.go` files. These files were utilized to test Modbus functionality within a controlled environment. In addition to these testing utilities, Mohammad contributed to parts of the `key-server.go` file.

Furthermore, Mohammad played a crucial role in writing the infrastructure tests for the project. He employed Ansible and Terraform to deploy Modware into a test environment, ensuring the reliability and functionality of the system. Mohammad also contributed extensively to the research paper, writing a majority of its sections and providing valuable insights into the overall project.

10.2 Chris Tremblay

Throughout the course of the project, I made significant contributions, taking the lead on various aspects of the project from start to finish. I was responsible for brainstorming the initial project idea, writing the project proposal, and conducting the preliminary literature review. In addition, I wrote the midterm update and implemented the entire Modware protocol in `Go`. The first iterations of the `mock_server.go`, and `modbus_client.go` files were initially implemented by me in `Python` too. I also conducted physical testing in a lab environment. I wrote the Background, Implementation Details, and Physical Results sections along with the initial versions of the Abstract, Introduction, Related Works, Project Idea, Challenge, and Future Work sections of the paper. I also provided a handful of references to my groupmate for Related Work and created the sequence diagrams to illustrate the data flow between the Modbus master and slave devices. All Physical Results screenshots were also created and annotated by me.

References

[1] AHN, B., BERE, G., AHMAD, S., CHOI, J., KIM, T., AND PARK, S.-W. Blockchain-enabled security module for transforming conventional inverters toward firmware security-enhanced smart inverters. In *2021 IEEE Energy Conversion Congress and Exposition (ECCE)* (2021), pp. 1307–1312.

[2] ALVES, THIAGO. Open-source plc software.

[3] ANDERSON, J., ALKABANI, Y., AND EL-GHAZAWI, T. Recpe: A pe for reconfigurable lightweight cryptography. In *2021 IEEE 34th International System-on-Chip Conference (SOCC)* (2021), pp. 176–181.

[4] CHEN, B., PATTANAIK, N., GOULART, A., BUTLER-PURRY, K. L., AND KUNDUR, D. Implementing attacks for modbus/tcp protocol in a real-time cyber physical system test bed. In *2015 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR)* (2015), pp. 1–6.

[5] DESRUISSEAUX, D. Modbus security – new protocol to improve control system security, Jul 2020.

[6] FOVINO, I. N., CARCANO, A., DE LACHEZE MUREL, T., TROMBETTA, A., AND MASERA, M. Modbus/dnp3 state-based intrusion detection system. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications* (2010), pp. 729–736.

[7] HUPP, W., HASANDKA, A., DE CARVALHO, R. S., AND SALEEM, D. Module-ot: A hardware security module for operational technology. In *2020 IEEE Texas Power and Energy Conference (TPEC)* (2020), pp. 1–6.

[8] INDUCTIVE AUTOMATION. Ignition.

[9] KANG, B., MAYNARD, P., MCLAUGHLIN, K., SEZER, S., ANDRÉN, F., SEITL, C., KUPZOG, F., AND STRASSER, T. Investigating cyber-physical attacks against iec 61850 photovoltaic inverter installations. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)* (2015), pp. 1–8.

[10] KROTOFIL, M., AND GOLLMANN, D. Industrial control systems security: What is happening? In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)* (2013), pp. 670–675.

[11] LI, B., CHEN, G., WANG, L., AND HAO, Z. Tower crane remote wireless monitoring system based on modbus/tcp protocol. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (2017), vol. 2, pp. 187–190.

[12] MORRIS, T., AND PAVURAPU, K. A retrofit network transaction data logger and intrusion detection system for transmission and distribution substations. In *2010 IEEE International Conference on Power and Energy* (2010), pp. 958–963.

[13] MORRIS, T. H., JONES, B. A., VAUGHN, R. B., AND DANDASS, Y. S. Deterministic intrusion detection rules for modbus protocols. In *2013 46th Hawaii International Conference on System Sciences* (2013), pp. 1773–1781.

[14] NARDONE, R., RODRÍGUEZ, R. J., AND MARRONE, S. Formal security assessment of modbus protocol. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)* (2016), pp. 142–147.

[15] OKABE, N., SAKANE, S., MIYAZAWA, K., KAMADA, K., INOUE, A., AND ISHIYAMA, M. Security architecture for control networks using ipsec and kink. In *The 2005 Symposium on Applications and the Internet* (2005), pp. 414–420.

[16] RADOGLOU-GRAMMATIKIS, P., SINIOSOGLOU, I., LIATIFIS, T., KOUROUNIADIS, A., ROMPOLOS, K., AND SARIGIANNIDIS, P. Implementation and detection of modbus cyberattacks. In *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)* (2020), pp. 1–4.

[17] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A Software-Only implementation of a TPM chip. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 841–856.

[18] RAVIKUMAR, G., SINGH, A., BABU, J. R., MOATAZ A, A., AND GOVINDARASU, M. D-ids for cyber-physical der modbus system - architecture, modeling, testbed-based evaluation. In *2020 Resilience Week (RWS)* (2020), pp. 153–159.

[19] ROMANOV, E. L., GERVAS, N. A., AND MENZHULIN, S. A. Platform-independent scada and iot architecture based on metadata. In *2022 IEEE International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)* (2022), pp. 1590–1595.

[20] SHANG, W., DING, Q., MARIANANTONI, A., BURKE, J., AND ZHANG, L. Securing building management systems using named data networking. *IEEE Network 28*, 3 (2014), 50–56.

[21] SRIDHAR, S., AND MANIMARAN, G. Data integrity attacks and their impacts on scada control system. In *IEEE PES General Meeting* (2010), pp. 1–6.

[22] TOC, S.-I., AND KORODI, A. Modbus-opc ua wrapper using node-red and iot-2040 with application in the water industry. In *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)* (2018), pp. 000099–000104.

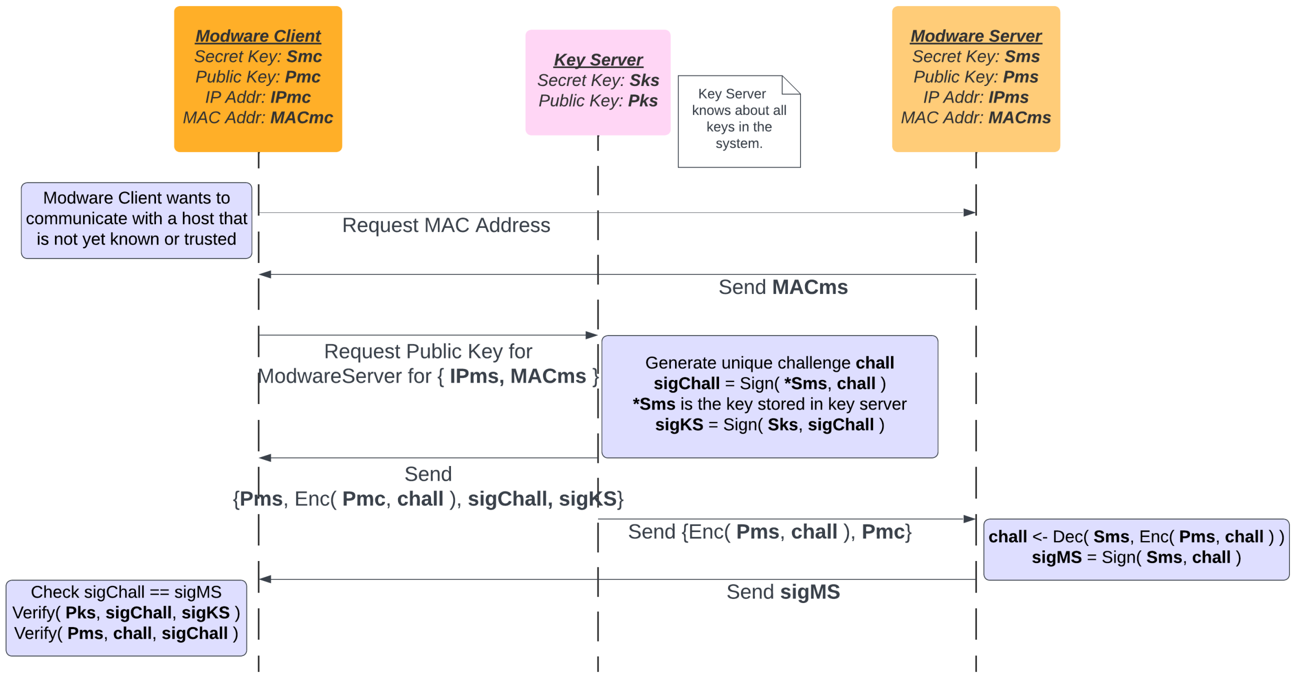
[23] WARREN, J. C. Ethernet/ip applications for electrical industrial systems. In *2009 IEEE Industry Applications Society Annual Meeting* (2009), pp. 1–5.

[24] WEI, C., XIJUN, W., WENXIA, S., AND RUITAO, Y. The design of profinet-modbus protocol conversion gateway based on the ertec 200p. In *2016 10th International Conference on Software, Knowledge, Information Management Applications (SKIMA)* (2016), pp. 87–91.

[25] WEST, A. Securing dnp3 and modbus with aga12-2j. In *2008 IEEE Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century* (2008), pp. 1–4.

[26] YOU, W., AND GE, H. Design and implementation of modbus protocol for intelligent building security. In *2019 IEEE 19th International Conference on Communication Technology (ICCT)* (2019), pp. 420–423.

Appendix A Protocol Sequence Diagrams



Modware Client
Secret Key: Smc
Public Key: Pmc
IP Addr: IPmc
MAC Addr: MACmc
Key Server
Secret Key: Sks
Public Key: Pks
Key Server knows about all keys in the system.
Modware Server
Secret Key: Sms
Public Key: Pms
IP Addr: IPms
MAC Addr: MACms
Modware Client wants to communicate with a host that is not yet known or trusted
Request MAC Address
Send MACms
Request Public Key for ModwareServer for { IPms, MACms }
Generate unique challenge chall
sigChall = Sign(*Sms, chall)
*Sms is the key stored in key server
sigKS = Sign(Sks, sigChall)
Send {Pms, Enc(Pmc, chall), sigChall, sigKS}
Send {Enc(Pms, chall), Pmc}
chall <- Dec(Sms, Enc(Pms, chall))
sigMS = Sign(Sms, chall)
Send sigMS
Check sigChall == sigMS
Verify(Pks, sigChall, sigKS)
Verify(Pms, chall, sigChall)

Figure 2: Verify Host Protocol Flow

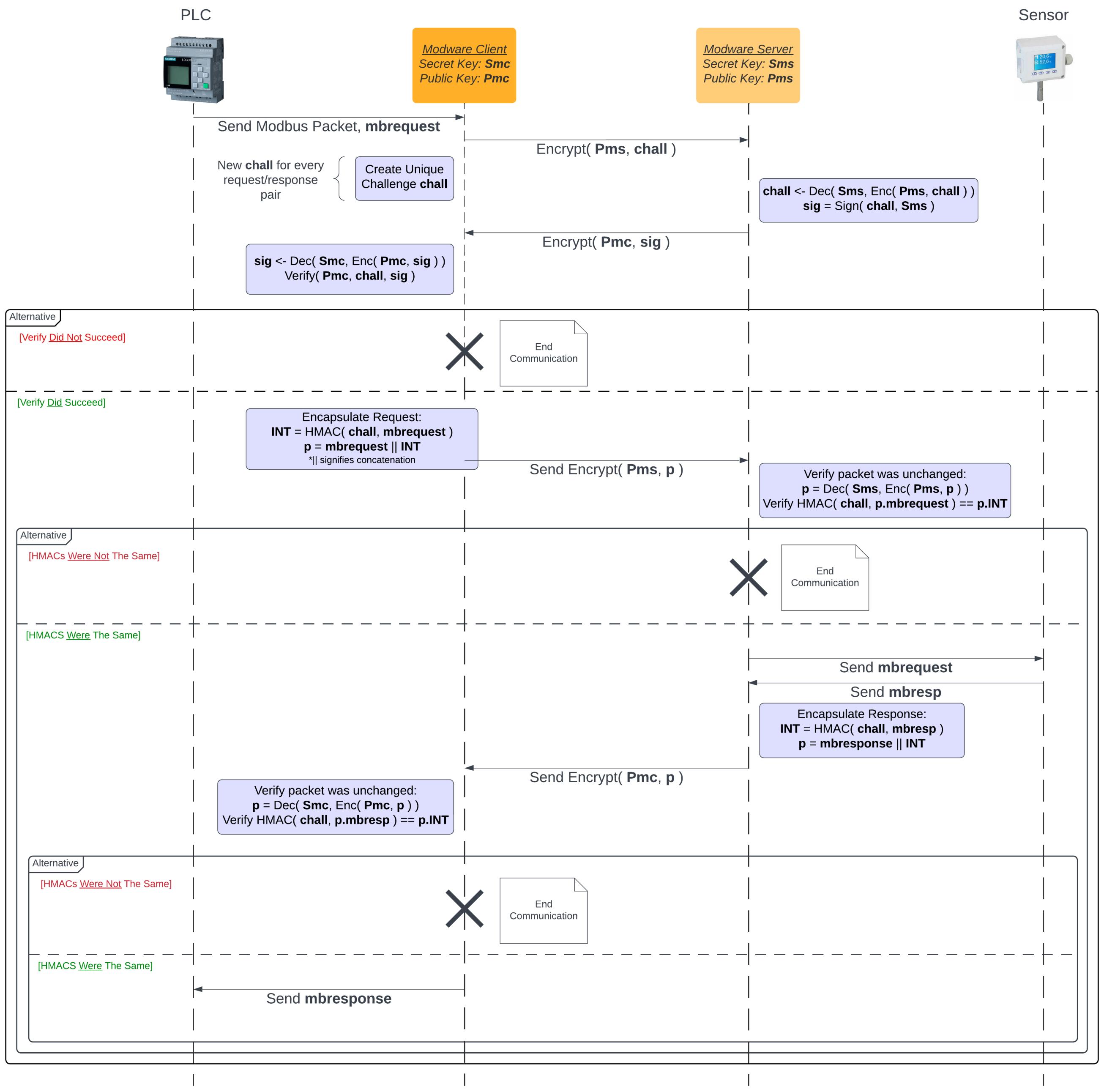

PLC
Modware Client
Secret Key: Smc
Public Key: Pmc
Modware Server
Secret Key: Sms
Public Key: Pms
Sensor
Send Modbus Packet, mbrequest
Encrypt(Pms, chall)
New chall for every request/response pair
Create Unique Challenge chall
chall <- Dec(Sms, Enc(Pms, chall))
sig = Sign(chall, Sms)
Encrypt(Pmc, sig)
sig <- Dec(Smc, Enc(Pmc, sig))
Verify(Pmc, chall, sig)
Alternative
[Verify Did Not Succeed]
End Communication
[Verify Did Succeed]
Encapsulate Request:
INT = HMAC(chall, mbrequest)
p = mbrequest || INT
*|| signifies concatenation
Send Encrypt(Pms, p)
Verify packet was unchanged:
p = Dec(Sms, Enc(Pms, p))
Verify HMAC(chall, p.mbrequest) == p.INT
Alternative
[HMACs Were Not The Same]
End Communication
[HMACS Were The Same]
Send mbrequest
Send mbresp
Encapsulate Response:
INT = HMAC(chall, mbresp)
p = mbresponse || INT
Send Encrypt(Pmc, p)
Verify packet was unchanged:
p = Dec(Smc, Enc(Pmc, p))
Verify HMAC(chall, p.mbresp) == p.INT
Alternative
[HMACs Were Not The Same]
End Communication
[HMACS Were The Same]
Send mbresponse

Figure 3: Communication Between Verified Hosts

Appendix B Physical Experimental Results Figures

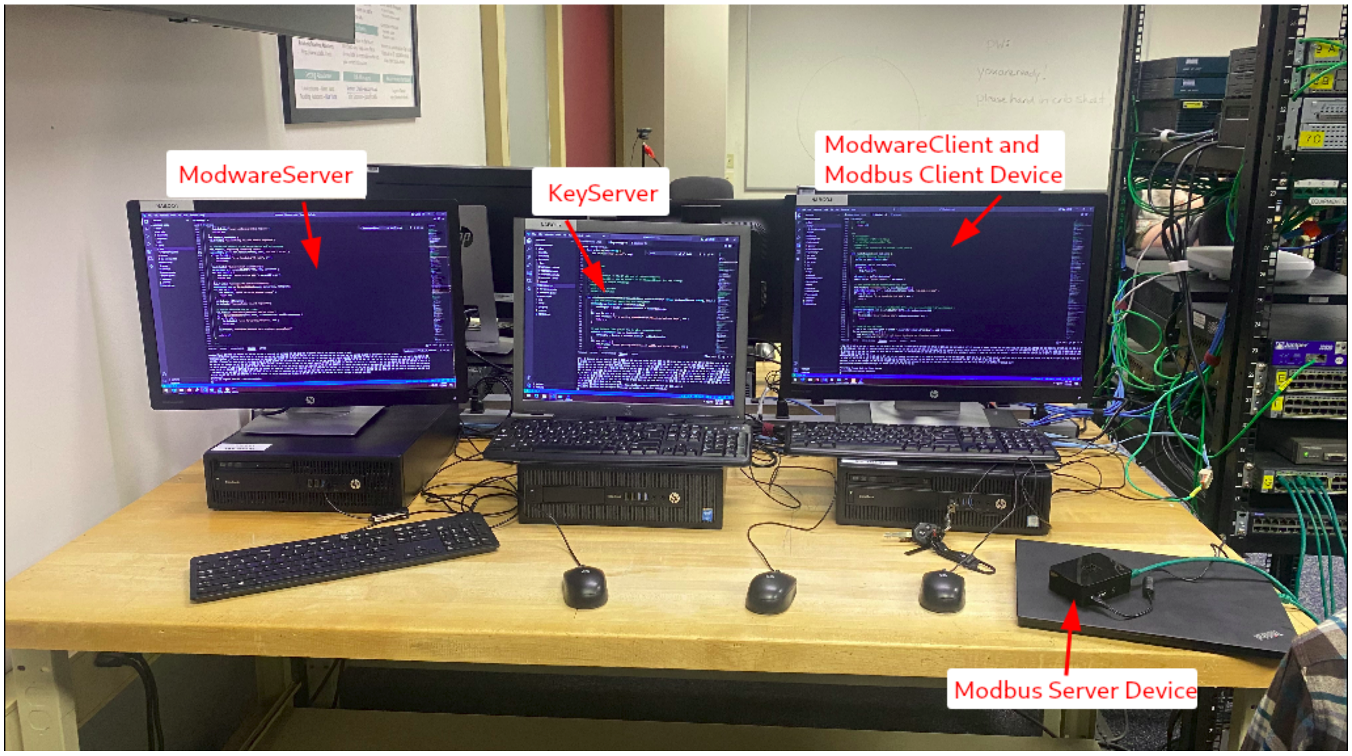
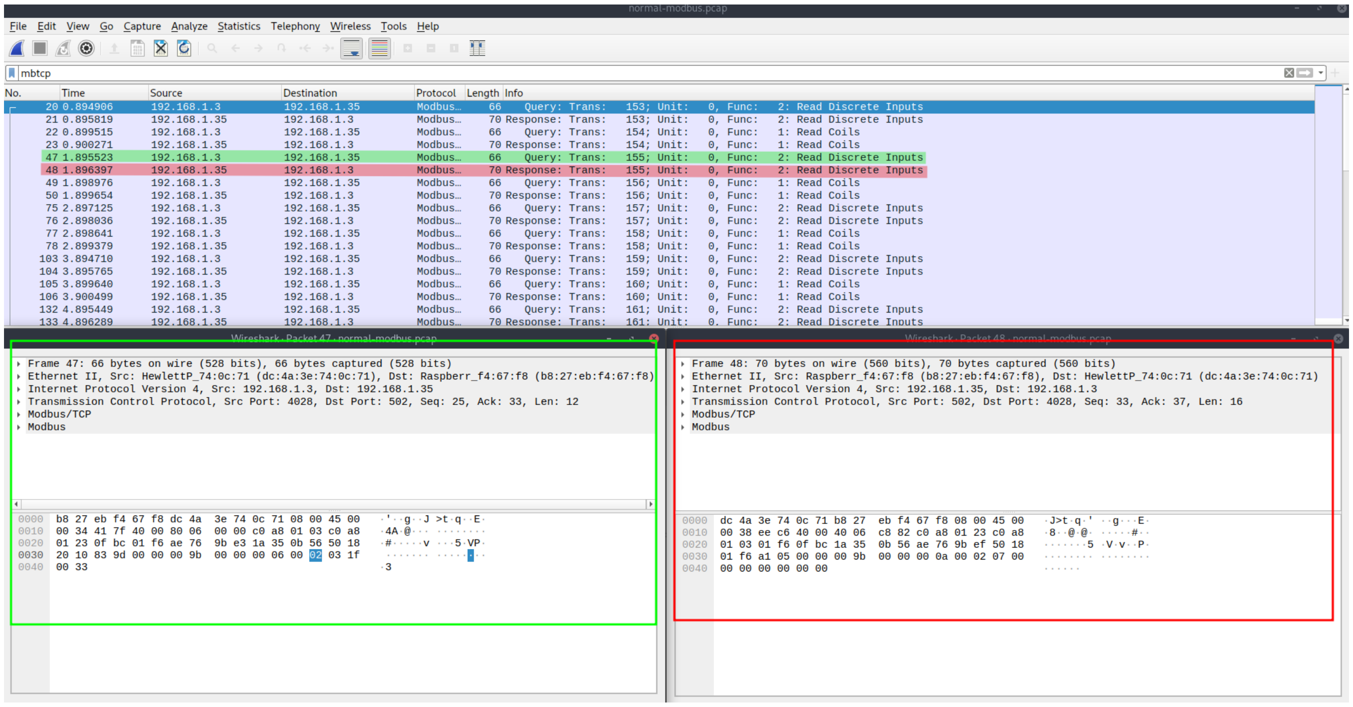


Figure 4: Lab Setup

Machine	IP	Description
Windows PC	192.168.1.2	ModwareServer
Windows PC	192.168.1.3	ModwareClient and Modbus Client Device
Windows PC	192.168.1.4	KeyServer
RaspberryPi	192.168.1.35	Modbus Server Device

Table 1: Description of Devices



```
mbtcp
No.  Time      Source        Destination   Protocol Length Info
 20 0.894906  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   153; Unit:   0, Func:   2: Read Discrete Inputs
 21 0.895819  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   153; Unit:   0, Func:   2: Read Discrete Inputs
 22 0.899515  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   154; Unit:   0, Func:   1: Read Coils
 23 0.900271  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   154; Unit:   0, Func:   1: Read Coils
 47 1.895523  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   155; Unit:   0, Func:   2: Read Discrete Inputs
 48 1.896397  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   155; Unit:   0, Func:   2: Read Discrete Inputs
 49 1.898976  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   156; Unit:   0, Func:   1: Read Coils
 50 1.899654  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   156; Unit:   0, Func:   1: Read Coils
 75 2.897125  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   157; Unit:   0, Func:   2: Read Discrete Inputs
 76 2.898036  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   157; Unit:   0, Func:   2: Read Discrete Inputs
 77 2.898641  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   158; Unit:   0, Func:   1: Read Coils
 78 2.899379  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   158; Unit:   0, Func:   1: Read Coils
103 3.894710  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   159; Unit:   0, Func:   2: Read Discrete Inputs
104 3.895765  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   159; Unit:   0, Func:   2: Read Discrete Inputs
105 3.899640  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   160; Unit:   0, Func:   1: Read Coils
106 3.900499  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   160; Unit:   0, Func:   1: Read Coils
132 4.895449  192.168.1.3   192.168.1.35  Modbus…  66     Query: Trans:   161; Unit:   0, Func:   2: Read Discrete Inputs
133 4.896289  192.168.1.35  192.168.1.3   Modbus…  70 Response: Trans:   161; Unit:   0, Func:   2: Read Discrete Inputs
```

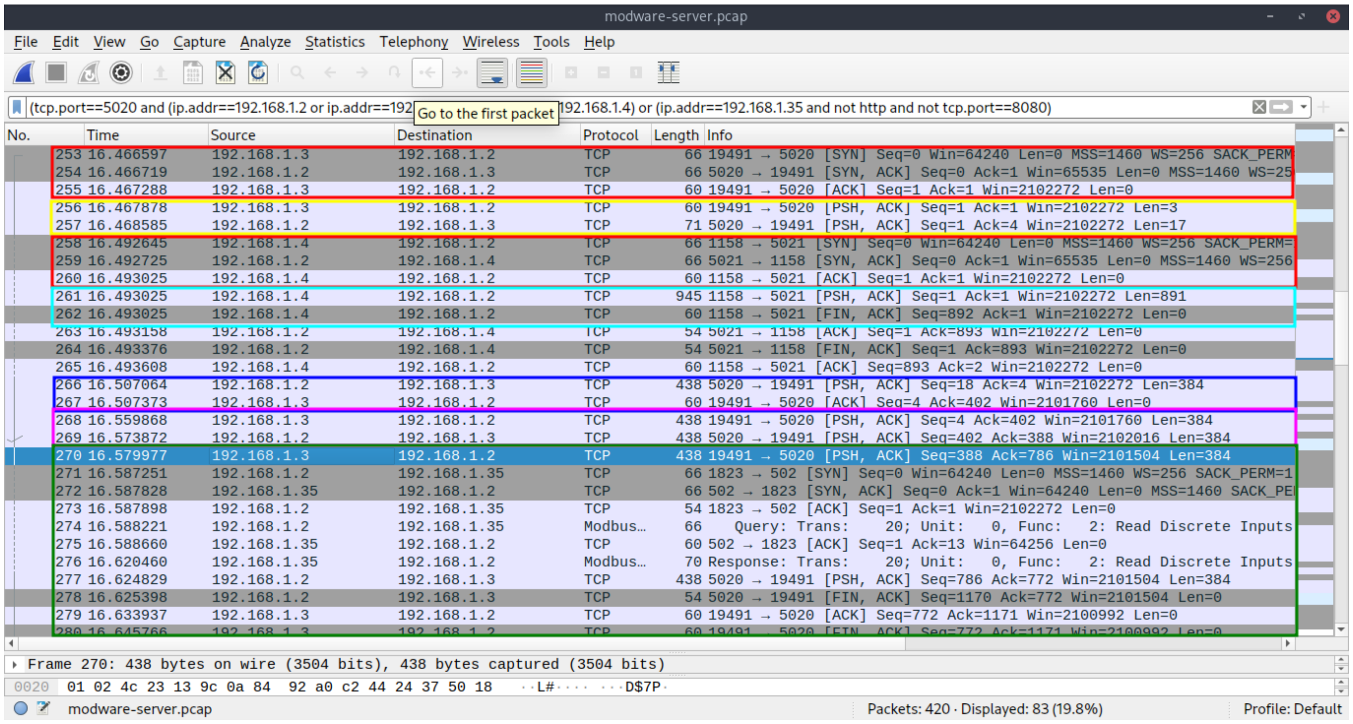
```
Wireshark · Packet 47 · normal-modbus.pcap
› Frame 47: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
› Ethernet II, Src: HewlettP_74:0c:71 (dc:4a:3e:74:0c:71), Dst: Raspberr_f4:67:f8 (b8:27:eb:f4:67:f8)
› Internet Protocol Version 4, Src: 192.168.1.3, Dst: 192.168.1.35
› Transmission Control Protocol, Src Port: 4028, Dst Port: 502, Seq: 25, Ack: 33, Len: 12
› Modbus/TCP
› Modbus

0000  b8 27 eb f4 67 f8 dc 4a  3e 74 0c 71 08 00 45 00   ·'··g··J >t·q··E·
0010  00 34 41 7f 40 00 80 06  00 00 c0 a8 01 03 c0 a8   ·4A·@··· ········
0020  01 23 0f bc 01 f6 ae 76  9b e3 1a 35 0b 56 50 18   ·#·····v ···5·VP·
0030  20 10 83 9d 00 00 00 9b  00 00 00 06 00 02 03 1f    ······· ·····.··
0040  00 33                                              ·3
```

```
Wireshark · Packet 48 · normal-modbus.pcap
› Frame 48: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
› Ethernet II, Src: Raspberr_f4:67:f8 (b8:27:eb:f4:67:f8), Dst: HewlettP_74:0c:71 (dc:4a:3e:74:0c:71)
› Internet Protocol Version 4, Src: 192.168.1.35, Dst: 192.168.1.3
› Transmission Control Protocol, Src Port: 502, Dst Port: 4028, Seq: 33, Ack: 37, Len: 16
› Modbus/TCP
› Modbus

0000  dc 4a 3e 74 0c 71 b8 27  eb f4 67 f8 08 00 45 00   ·J>t·q·' ··g···E·
0010  00 38 ee c6 40 00 40 06  c8 82 c0 a8 01 23 c0 a8   ·8··@·@· ·····#··
0020  01 03 01 f6 0f bc 1a 35  0b 56 ae 76 9b ef 50 18   ·······5 ·V·v··P·
0030  01 f6 a1 05 00 00 00 9b  00 00 00 0a 00 02 07 00   ········ ········
0040  00 00 00 00 00 00                                  ······
```

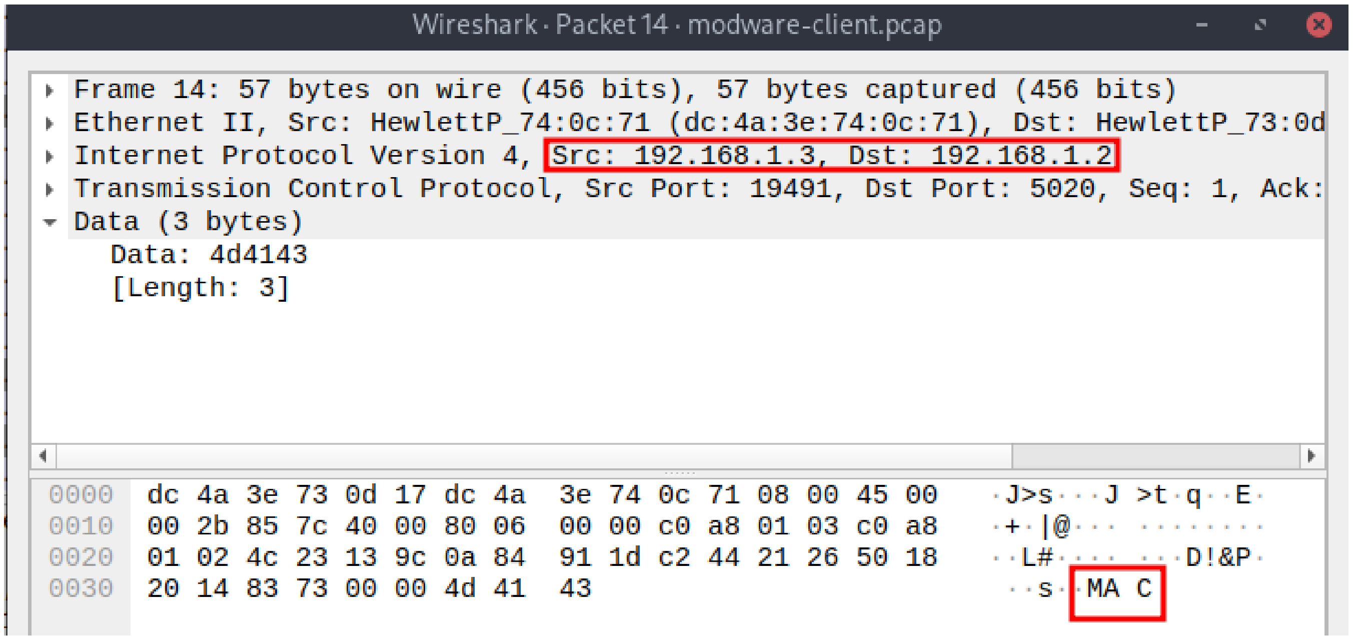
Figure 5: Normal Modbus Traffic. Green: Modbus Query, Red: Modbus Response



```
(tcp.port==5020 and (ip.addr==192.168.1.2 or ip.addr==192 Go to the first packet 192.168.1.4) or (ip.addr==192.168.1.35 and not http and not tcp.port==8080)
No.  Time       Source        Destination   Protocol Length Info
253 16.466597  192.168.1.3   192.168.1.2   TCP         66 19491 → 5020 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
254 16.466719  192.168.1.2   192.168.1.3   TCP         66 5020 → 19491 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=25
255 16.467288  192.168.1.3   192.168.1.2   TCP         60 19491 → 5020 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
256 16.467878  192.168.1.3   192.168.1.2   TCP         60 19491 → 5020 [PSH, ACK] Seq=1 Ack=1 Win=2102272 Len=3
257 16.468585  192.168.1.2   192.168.1.3   TCP         71 5020 → 19491 [PSH, ACK] Seq=1 Ack=4 Win=2102272 Len=17
258 16.492645  192.168.1.4   192.168.1.2   TCP         66 1158 → 5021 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=
259 16.492725  192.168.1.2   192.168.1.4   TCP         66 5021 → 1158 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256
260 16.493025  192.168.1.4   192.168.1.2   TCP         60 1158 → 5021 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
261 16.493025  192.168.1.4   192.168.1.2   TCP        945 1158 → 5021 [PSH, ACK] Seq=1 Ack=1 Win=2102272 Len=891
262 16.493025  192.168.1.4   192.168.1.2   TCP         60 1158 → 5021 [FIN, ACK] Seq=892 Ack=1 Win=2102272 Len=0
263 16.493158  192.168.1.2   192.168.1.4   TCP         54 5021 → 1158 [ACK] Seq=1 Ack=893 Win=2102272 Len=0
264 16.493376  192.168.1.2   192.168.1.4   TCP         54 5021 → 1158 [FIN, ACK] Seq=1 Ack=893 Win=2102272 Len=0
265 16.493608  192.168.1.4   192.168.1.2   TCP         60 1158 → 5021 [ACK] Seq=893 Ack=2 Win=2102272 Len=0
266 16.507064  192.168.1.2   192.168.1.3   TCP        438 5020 → 19491 [PSH, ACK] Seq=18 Ack=4 Win=2102272 Len=384
267 16.507373  192.168.1.3   192.168.1.2   TCP         60 19491 → 5020 [ACK] Seq=4 Ack=402 Win=2101760 Len=0
268 16.559868  192.168.1.3   192.168.1.2   TCP        438 19491 → 5020 [PSH, ACK] Seq=4 Ack=402 Win=2101760 Len=384
269 16.573872  192.168.1.2   192.168.1.3   TCP        438 5020 → 19491 [PSH, ACK] Seq=402 Ack=388 Win=2102016 Len=384
270 16.579977  192.168.1.3   192.168.1.2   TCP        438 19491 → 5020 [PSH, ACK] Seq=388 Ack=786 Win=2101504 Len=384
271 16.587251  192.168.1.2   192.168.1.35  TCP         66 1823 → 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
272 16.587828  192.168.1.35  192.168.1.2   TCP         66 502 → 1823 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PE
273 16.587898  192.168.1.2   192.168.1.35  TCP         54 1823 → 502 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
274 16.588221  192.168.1.2   192.168.1.35  Modbus…     66    Query: Trans:    20; Unit:   0, Func:   2: Read Discrete Inputs
275 16.588660  192.168.1.35  192.168.1.2   TCP         60 502 → 1823 [ACK] Seq=1 Ack=13 Win=64256 Len=0
276 16.620460  192.168.1.35  192.168.1.2   Modbus…     70 Response: Trans:    20; Unit:   0, Func:   2: Read Discrete Inputs
277 16.624829  192.168.1.2   192.168.1.3   TCP        438 5020 → 19491 [PSH, ACK] Seq=786 Ack=772 Win=2101504 Len=384
278 16.625398  192.168.1.2   192.168.1.3   TCP         54 5020 → 19491 [FIN, ACK] Seq=1170 Ack=772 Win=2101504 Len=0
279 16.633937  192.168.1.3   192.168.1.2   TCP         60 19491 → 5020 [ACK] Seq=772 Ack=1171 Win=2100992 Len=0
280 16.645766  192.168.1.3   192.168.1.2   TCP         60 19491 → 5020 [FIN, ACK] Seq=772 Ack=1171 Win=2100992 Len=0

› Frame 270: 438 bytes on wire (3504 bits), 438 bytes captured (3504 bits)
0020  01 02 4c 23 13 9c 0a 84  92 a0 c2 44 24 37 50 18   ··L#···· ···D$7P·
modware-server.pcap                    Packets: 420 · Displayed: 83 (19.8%)        Profile: Default
```

Figure 6: Modware Protocol Flow, ModwareServer Wireshark Capture

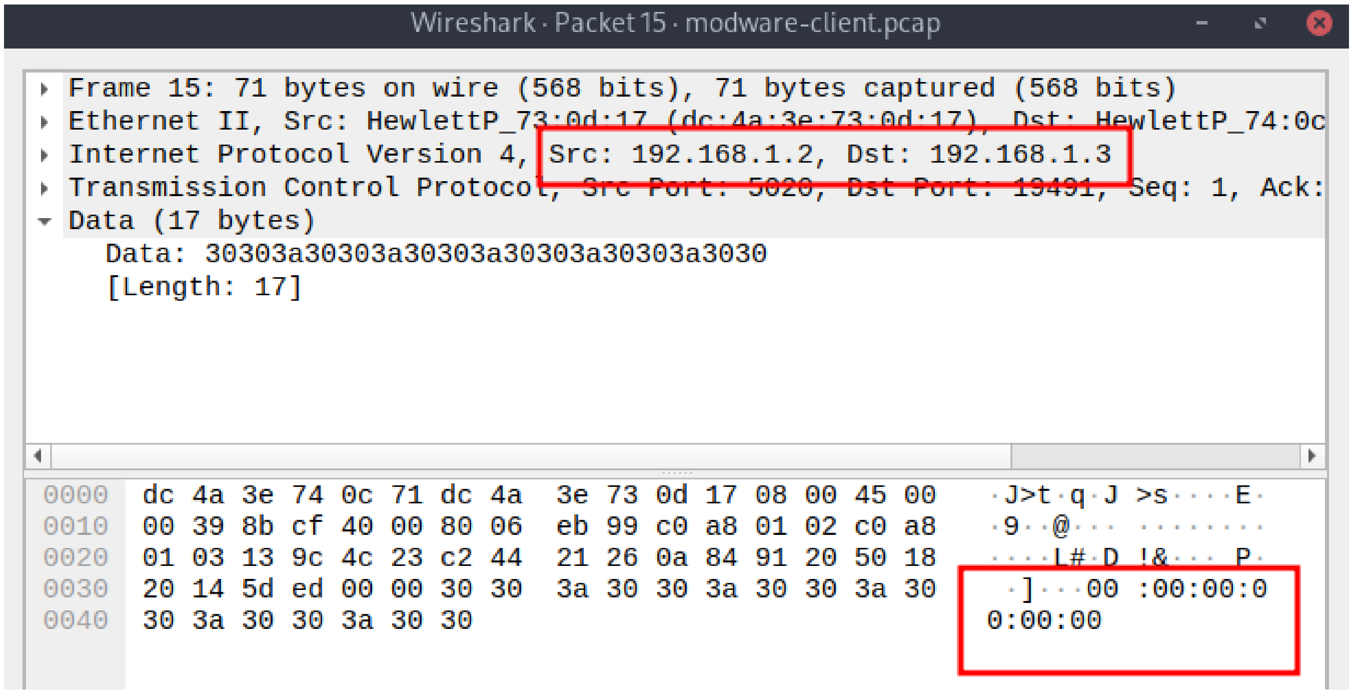


```
Wireshark · Packet 14 · modware-client.pcap

▸ Frame 14: 57 bytes on wire (456 bits), 57 bytes captured (456 bits)
▸ Ethernet II, Src: HewlettP_74:0c:71 (dc:4a:3e:74:0c:71), Dst: HewlettP_73:0d
▸ Internet Protocol Version 4, Src: 192.168.1.3, Dst: 192.168.1.2
▸ Transmission Control Protocol, Src Port: 19491, Dst Port: 5020, Seq: 1, Ack:
▾ Data (3 bytes)
    Data: 4d4143
    [Length: 3]

0000  dc 4a 3e 73 0d 17 dc 4a  3e 74 0c 71 08 00 45 00   ·J>s···J >t·q··E·
0010  00 2b 85 7c 40 00 80 06  00 00 c0 a8 01 03 c0 a8   ·+·|@··· ········
0020  01 02 4c 23 13 9c 0a 84  91 1d c2 44 21 26 50 18   ··L#···· ···D!&P·
0030  20 14 83 73 00 00 4d 41  43                         ··s··MA C
```

Figure 7: ModwareClient to ModwareServer MAC Request *VerifyHost*

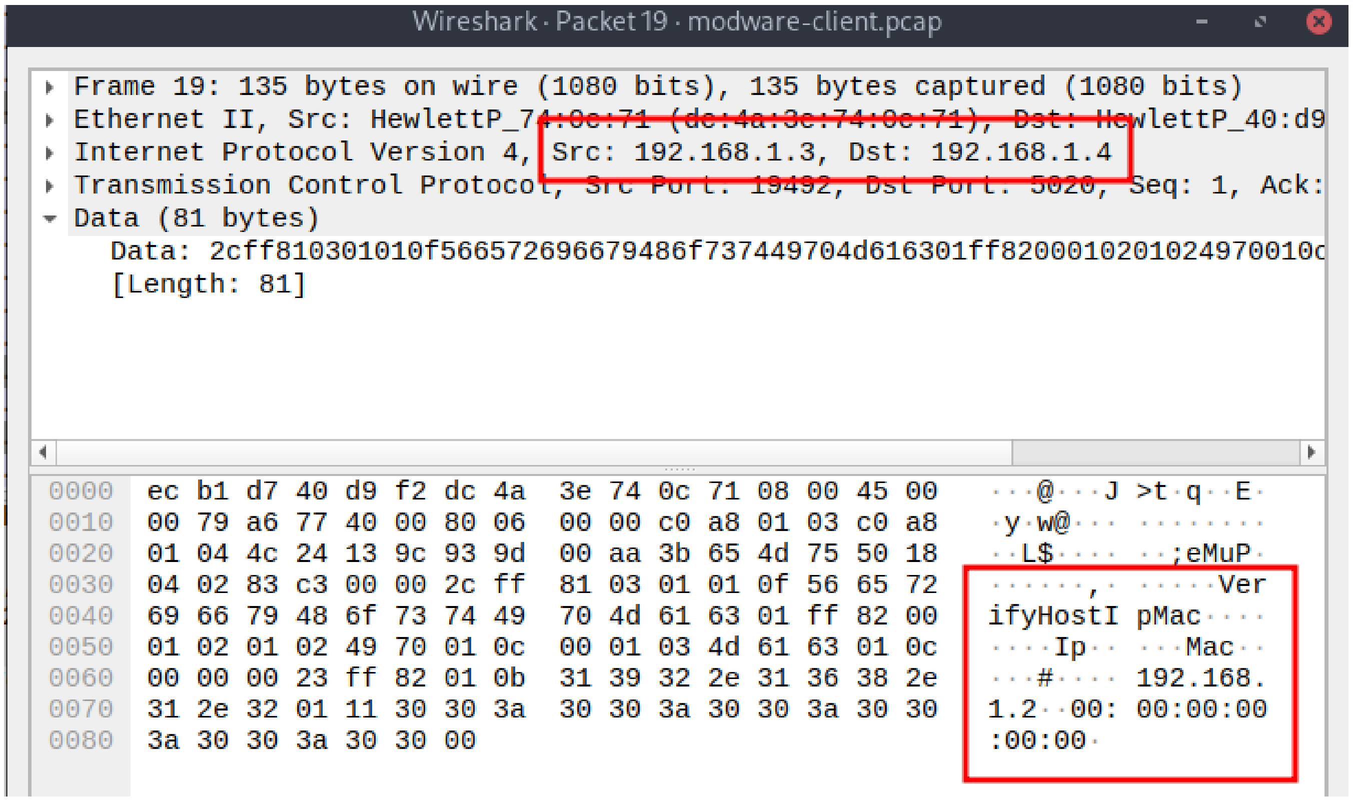


```
Wireshark · Packet 15 · modware-client.pcap

▸ Frame 15: 71 bytes on wire (568 bits), 71 bytes captured (568 bits)
▸ Ethernet II, Src: HewlettP_73:0d:17 (dc:4a:3e:73:0d:17), Dst: HewlettP_74:0c
▸ Internet Protocol Version 4, Src: 192.168.1.2, Dst: 192.168.1.3
▸ Transmission Control Protocol, Src Port: 5020, Dst Port: 19491, Seq: 1, Ack:
▾ Data (17 bytes)
    Data: 30303a30303a30303a30303a30303a3030
    [Length: 17]

0000  dc 4a 3e 74 0c 71 dc 4a  3e 73 0d 17 08 00 45 00   ·J>t·q·J >s····E·
0010  00 39 8b cf 40 00 80 06  eb 99 c0 a8 01 02 c0 a8   ·9··@··· ········
0020  01 03 13 9c 4c 23 c2 44  21 26 0a 84 91 20 50 18   ····L#·D !&··· P·
0030  20 14 5d ed 00 00 30 30  3a 30 30 3a 30 30 3a 30    ·]···00 :00:00:0
0040  30 3a 30 30 3a 30 30                               0:00:00
```

Figure 8: ModwareServer to ModwareClinet MAC Response *VerifyHost*

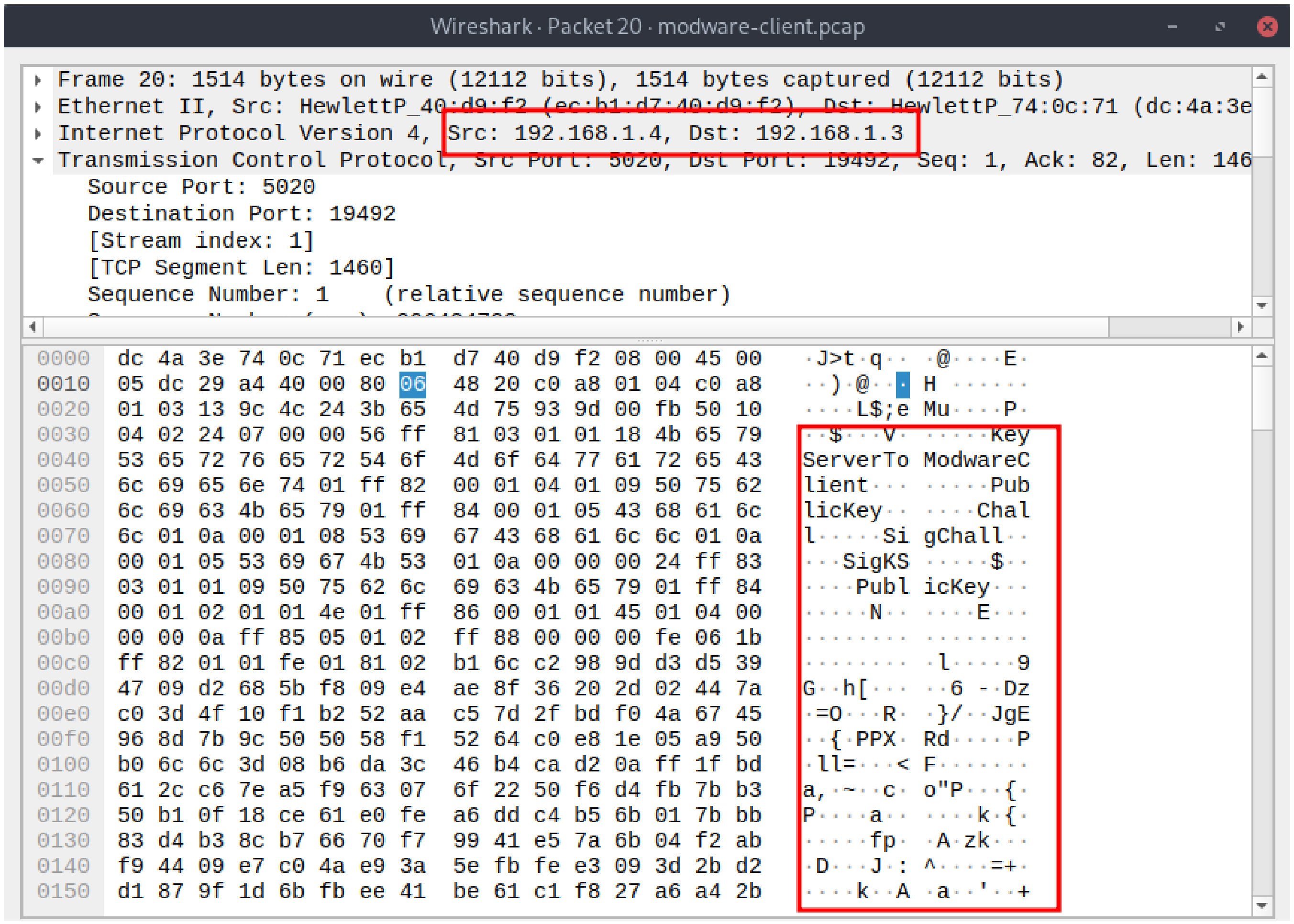


Wireshark · Packet 19 · modware-client.pcap

```
Frame 19: 135 bytes on wire (1080 bits), 135 bytes captured (1080 bits)
Ethernet II, Src: HewlettP_74:0c:71 (dc:4a:3e:74:0c:71), Dst: HewlettP_40:d9
Internet Protocol Version 4, Src: 192.168.1.3, Dst: 192.168.1.4
Transmission Control Protocol, Src Port: 19492, Dst Port: 5020, Seq: 1, Ack:
Data (81 bytes)
   Data: 2cff810301010f566572696679486f737449704d616301ff820001020102497001 0c
   [Length: 81]
```

```
0000  ec b1 d7 40 d9 f2 dc 4a  3e 74 0c 71 08 00 45 00   ···@···J >t·q··E·
0010  00 79 a6 77 40 00 80 06  00 00 c0 a8 01 03 c0 a8   ·y·w@··· ········
0020  01 04 4c 24 13 9c 93 9d  00 aa 3b 65 4d 75 50 18   ··L$···· ··;eMuP·
0030  04 02 83 c3 00 00 2c ff  81 03 01 01 0f 56 65 72   ······,· ·····Ver
0040  69 66 79 48 6f 73 74 49  70 4d 61 63 01 ff 82 00   ifyHostI pMac····
0050  01 02 01 02 49 70 01 0c  00 01 03 4d 61 63 01 0c   ····Ip·· ···Mac··
0060  00 00 00 23 ff 82 01 0b  31 39 32 2e 31 36 38 2e   ···#···· 192.168.
0070  31 2e 32 01 11 30 30 3a  30 30 3a 30 30 3a 30 30   1.2··00: 00:00:00
0080  3a 30 30 3a 30 30 00                               :00:00·
```

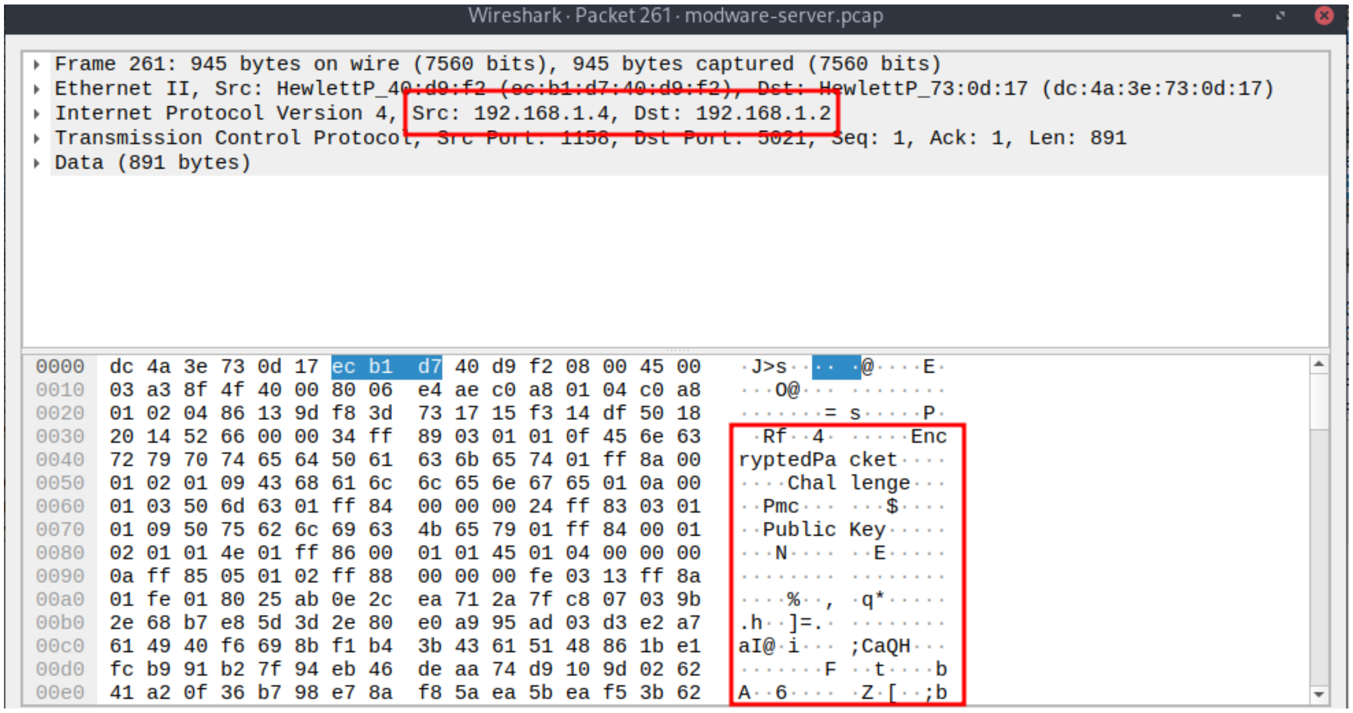
Figure 9: ModwareClient to KeyServer Public Key Request *VerifyHost*


Wireshark · Packet 20 · modware-client.pcap

```
▸ Frame 20: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
▸ Ethernet II, Src: HewlettP_40:d9:f2 (ec:b1:d7:40:d9:f2), Dst: HewlettP_74:0c:71 (dc:4a:3e
▸ Internet Protocol Version 4, Src: 192.168.1.4, Dst: 192.168.1.3
▾ Transmission Control Protocol, Src Port: 5020, Dst Port: 19492, Seq: 1, Ack: 82, Len: 146
    Source Port: 5020
    Destination Port: 19492
    [Stream index: 1]
    [TCP Segment Len: 1460]
    Sequence Number: 1    (relative sequence number)
```

```
0000  dc 4a 3e 74 0c 71 ec b1  d7 40 d9 f2 08 00 45 00   ·J>t·q·· ·@····E·
0010  05 dc 29 a4 40 00 80 06  48 20 c0 a8 01 04 c0 a8   ··)·@··· H ······
0020  01 03 13 9c 4c 24 3b 65  4d 75 93 9d 00 fb 50 10   ····L$;e Mu····P·
0030  04 02 24 07 00 00 56 ff  81 03 01 01 18 4b 65 79   ··$···V· ·····Key
0040  53 65 72 76 65 72 54 6f  4d 6f 64 77 61 72 65 43   ServerTo ModwareC
0050  6c 69 65 6e 74 01 ff 82  00 01 04 01 09 50 75 62   lient··· ·····Pub
0060  6c 69 63 4b 65 79 01 ff  84 00 01 05 43 68 61 6c   licKey·· ····Chal
0070  6c 01 0a 00 01 08 53 69  67 43 68 61 6c 6c 01 0a   l·····Si gChall··
0080  00 01 05 53 69 67 4b 53  01 0a 00 00 00 24 ff 83   ···SigKS ·····$··
0090  03 01 01 09 50 75 62 6c  69 63 4b 65 79 01 ff 84   ····Publ icKey···
00a0  00 01 02 01 01 4e 01 ff  86 00 01 01 45 01 04 00   ·····N·· ····E···
00b0  00 00 0a ff 85 05 01 02  ff 88 00 00 00 fe 06 1b   ········ ········
00c0  ff 82 01 01 fe 01 81 02  b1 6c c2 98 9d d3 d5 39   ········ ·l·····9
00d0  47 09 d2 68 5b f8 09 e4  ae 8f 36 20 2d 02 44 7a   G··h[··· ··6 -·Dz
00e0  c0 3d 4f 10 f1 b2 52 aa  c5 7d 2f bd f0 4a 67 45   ·=O···R· ·}/··JgE
00f0  96 8d 7b 9c 50 50 58 f1  52 64 c0 e8 1e 05 a9 50   ··{·PPX· Rd·····P
0100  b0 6c 6c 3d 08 b6 da 3c  46 b4 ca d2 0a ff 1f bd   ·ll=···< F·······
0110  61 2c c6 7e a5 f9 63 07  6f 22 50 f6 d4 fb 7b b3   a,·~··c· o"P···{·
0120  50 b1 0f 18 ce 61 e0 fe  a6 dd c4 b5 6b 01 7b bb   P····a·· ····k·{·
0130  83 d4 b3 8c b7 66 70 f7  99 41 e5 7a 6b 04 f2 ab   ·····fp· ·A·zk···
0140  f9 44 09 e7 c0 4a e9 3a  5e fb fe e3 09 3d 2b d2   ·D···J·: ^····=+·
0150  d1 87 9f 1d 6b fb ee 41  be 61 c1 f8 27 a6 a4 2b   ····k··A ·a··'··+
```

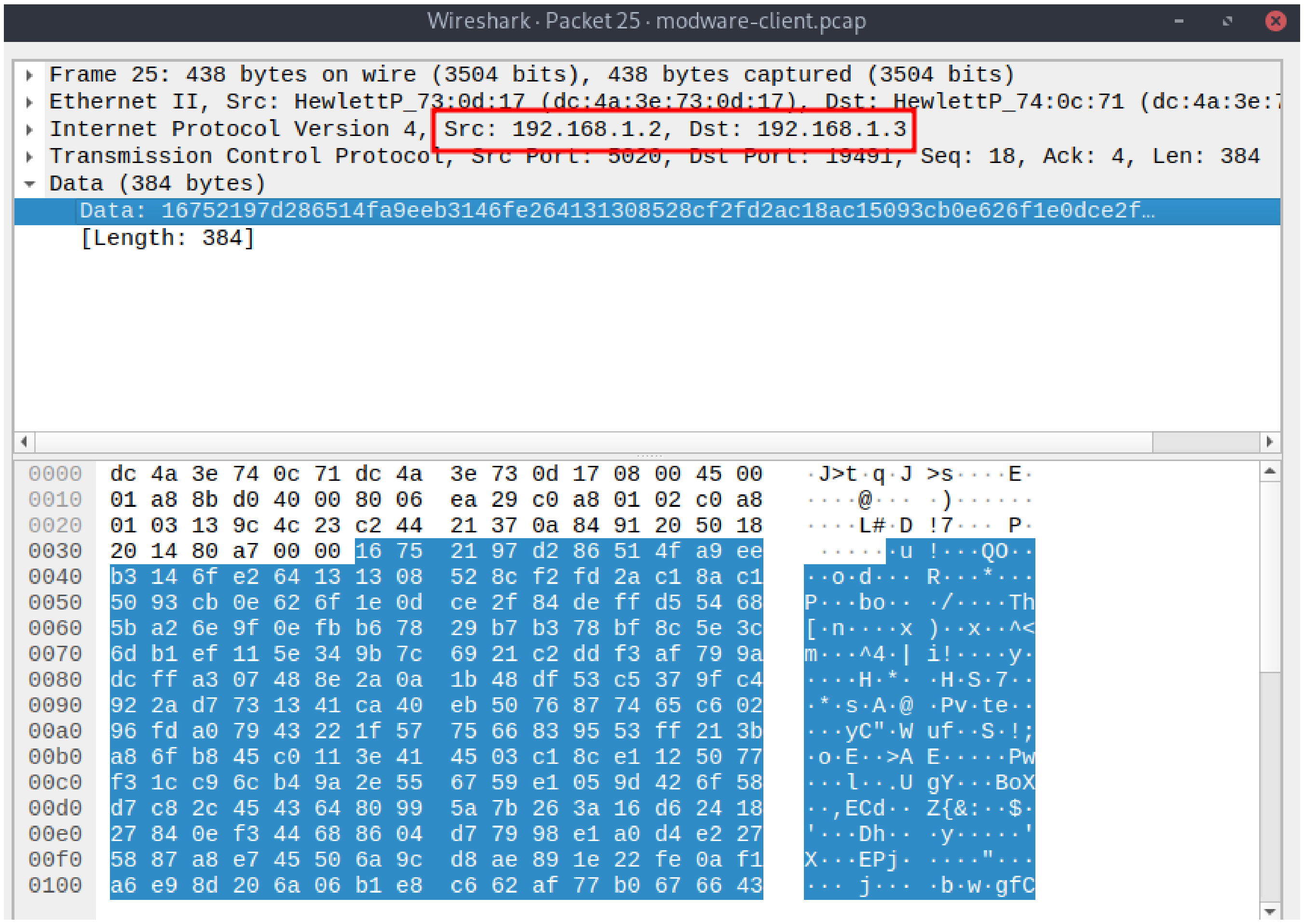
Figure 10: KeyServer to ModwareClient Public Key Response *VerifyHost*



```
▸ Frame 261: 945 bytes on wire (7560 bits), 945 bytes captured (7560 bits)
▸ Ethernet II, Src: HewlettP_40:d9:f2 (ec:b1:d7:40:d9:f2), Dst: HewlettP_73:0d:17 (dc:4a:3e:73:0d:17)
▸ Internet Protocol Version 4, Src: 192.168.1.4, Dst: 192.168.1.2
▸ Transmission Control Protocol, Src Port: 1158, Dst Port: 5021, Seq: 1, Ack: 1, Len: 891
▸ Data (891 bytes)
```

```
0000  dc 4a 3e 73 0d 17 ec b1  d7 40 d9 f2 08 00 45 00   ·J>s···· ·@····E·
0010  03 a3 8f 4f 40 00 80 06  e4 ae c0 a8 01 04 c0 a8   ···O@··· ········
0020  01 02 04 86 13 9d f8 3d  73 17 15 f3 14 df 50 18   ·······= s·····P·
0030  20 14 52 66 00 00 34 ff  89 03 01 01 0f 45 6e 63    ·Rf··4· ·····Enc
0040  72 79 70 74 65 64 50 61  63 6b 65 74 01 ff 8a 00   ryptedPa cket····
0050  01 02 01 09 43 68 61 6c  6c 65 6e 67 65 01 0a 00   ····Chal lenge···
0060  01 03 50 6d 63 01 ff 84  00 00 00 24 ff 83 03 01   ··Pmc··· ···$····
0070  01 09 50 75 62 6c 69 63  4b 65 79 01 ff 84 00 01   ··Public Key·····
0080  02 01 01 4e 01 ff 86 00  01 01 45 01 04 00 00 00   ···N···· ··E·····
0090  0a ff 85 05 01 02 ff 88  00 00 00 fe 03 13 ff 8a   ········ ········
00a0  01 fe 01 80 25 ab 0e 2c  ea 71 2a 7f c8 07 03 9b   ····%··, ·q*·····
00b0  2e 68 b7 e8 5d 3d 2e 80  e0 a9 95 ad 03 d3 e2 a7   .h··]=.· ········
00c0  61 49 40 f6 69 8b f1 b4  3b 43 61 51 48 86 1b e1   aI@·i··· ;CaQH···
00d0  fc b9 91 b2 7f 94 eb 46  de aa 74 d9 10 9d 02 62   ·······F ··t····b
00e0  41 a2 0f 36 b7 98 e7 8a  f8 5a ea 5b ea f5 3b 62   A··6···· ·Z·[··;b
```

Figure 11: KeyServer to Modware Server Public Key Response for *VerifyHost*



```
Wireshark · Packet 25 · modware-client.pcap

▸ Frame 25: 438 bytes on wire (3504 bits), 438 bytes captured (3504 bits)
▸ Ethernet II, Src: HewlettP_73:0d:17 (dc:4a:3e:73:0d:17), Dst: HewlettP_74:0c:71 (dc:4a:3e:7
▸ Internet Protocol Version 4, Src: 192.168.1.2, Dst: 192.168.1.3
▸ Transmission Control Protocol, Src Port: 5020, Dst Port: 19491, Seq: 18, Ack: 4, Len: 384
▾ Data (384 bytes)
    Data: 16752197d286514fa9eeb3146fe264131308528cf2fd2ac18ac15093cb0e626f1e0dce2f…
    [Length: 384]

0000  dc 4a 3e 74 0c 71 dc 4a  3e 73 0d 17 08 00 45 00   ·J>t·q·J >s····E·
0010  01 a8 8b d0 40 00 80 06  ea 29 c0 a8 01 02 c0 a8   ····@··· ·)······
0020  01 03 13 9c 4c 23 c2 44  21 37 0a 84 91 20 50 18   ····L#·D !7··· P·
0030  20 14 80 a7 00 00 16 75  21 97 d2 86 51 4f a9 ee    ······u !···QO··
0040  b3 14 6f e2 64 13 13 08  52 8c f2 fd 2a c1 8a c1   ··o·d··· R···*···
0050  50 93 cb 0e 62 6f 1e 0d  ce 2f 84 de ff d5 54 68   P···bo·· ·/····Th
0060  5b a2 6e 9f 0e fb b6 78  29 b7 b3 78 bf 8c 5e 3c   [·n····x )··x··^<
0070  6d b1 ef 11 5e 34 9b 7c  69 21 c2 dd f3 af 79 9a   m···^4·| i!····y·
0080  dc ff a3 07 48 8e 2a 0a  1b 48 df 53 c5 37 9f c4   ····H·*· ·H·S·7··
0090  92 2a d7 73 13 41 ca 40  eb 50 76 87 74 65 c6 02   ·*·s·A·@ ·Pv·te··
00a0  96 fd a0 79 43 22 1f 57  75 66 83 95 53 ff 21 3b   ···yC"·W uf··S·!;
00b0  a8 6f b8 45 c0 11 3e 41  45 03 c1 8c e1 12 50 77   ·o·E··>A E·····Pw
00c0  f3 1c c9 6c b4 9a 2e 55  67 59 e1 05 9d 42 6f 58   ···l··.U gY···BoX
00d0  d7 c8 2c 45 43 64 80 99  5a 7b 26 3a 16 d6 24 18   ··,ECd·· Z{&:··$·
00e0  27 84 0e f3 44 68 86 04  d7 79 98 e1 a0 d4 e2 27   '···Dh·· ·y·····'
00f0  58 87 a8 e7 45 50 6a 9c  d8 ae 89 1e 22 fe 0a f1   X···EPj· ····"···
0100  a6 e9 8d 20 6a 06 b1 e8  c6 62 af 77 b0 67 66 43   ··· j··· ·b·w·gfC
```

Figure 12: ModwareServer to ModwareClient Signed Challenge for *VerifyHost*