List of Software and Packages with versions:

1) Python:
    a) Device 1: Python 3.11.9
    b) Device 2: Python 3.12.3
2) Wireshark:
    Device 1: 4.0.8
    Device 2: 4.2.5
3) AVISPA:
    a) Device 2: 1.1
4) Libraries:
    a) socket
        i) Allows for communication between devices via endpoints known as sockets
        ii) Used to create network sockets for digital and physical twins
    b) threading
        i) Allows for thread management
        ii) Used to handle connections between devices
    c) random
        i) Allows for random number generation
        ii) Used to generate random computational values
    d) time
        i) Allows for work with dates, times, and timestamps
        ii) Used to take timestamps
    e) json
        i) Allows for functionality with JavaScript Object Notation data (JSON)
        ii) Used to convert dictionaries into strings and send data in a lightweight format
    f) tinyec
        i) Allows for elliptic curve cryptography
        ii) Used to generate keys
    g) secrets
        i) Allows for cryptographically strong random numbers
        ii) Used to select a cryptographically strong number on the elliptic curve
    h) hashlib
        i) Allows for use of hashing algorithms
        ii) Used to implement the SHA-256 and SHA-512 algorithms
    i) Crypto
        i) Allows for cryptographic primitives use
        ii) Used to implement AES encryption/decryption and random prime number generation
    j) os
        i) Allows for interaction with operating systems
        ii) Used to automate the authentication process
    k) Requests

i) Used in making HTTP requests
ii) Allows for the automation python files to retrieve the raw data from the github of the protocol to run locally on your machine

Installation of Software and Packages:

a) Windows
   i) Step 1: Install IDE (IDE used: Visual Studio Code)
      1) Download Visual Studio Code link
   ii) Step 2: Install Python
      1) Download Python link
      2) pip install <insert library> (socket, threading, random, time, json, tinyec, secrets, hashlib, Crypto, os)

b) MAC
   i) Step 1: Install IDE (IDE used: Visual Studio Code)
      1) Download Visual Studio Code link
   ii) Step 2: Install Python
      1) Download Python link
      2) pip3 install <insert library> (socket, threading, random, time, json, tinyec, secrets, hashlib, Crypto, os)

c) Linux
   i) Step 1: Install IDE (IDE used: Visual Studio Code)
      1) Download Visual Studio Code link
   ii) Step 2: Install Python
      1) Download Python link
      2) pip3 install <insert library> (socket, threading, random, time, json, tinyec, secrets, hashlib, Crypto, os)

d) Wireshark
   i) Step 1: Install Wireshark
      1) Download Wireshark link

GitHub link:
https://github.com/jwwyeth/REU-Data-Transfer-in-Smart-Environment/tree/main
Execution video (screencast) link:

Description of files in GitHub:

RE.py:

RE.py serves as the registry enroller for the protocol and allocates the starting set of values to AMDT1, AMDT2, AM1, and AM2 to then be used in DTSMAKA-1 and DTSMAKA-2. These values given are the pseudo-identity (RID), temporal identity (TID), g, p, q, private key, and public key. The g, p, and q represent random computational values. After the value set is given, the socket between RE and any of the other files is broken and the file is then ready to begin authentication.

AMDT1.py:

AMDT1.py emulates AMDT1, a digital twin in the protocol. It is given its starting values from the registry enroller and initiates DSTMAKA-1 by authentication with AMDT2.py to produce a shared session key. Afterwards, AMDT1.py starts DSTMAKA-2 by authenticating AM1.py to produce their separately distinct session key. The .gcode and .ngc files are then read by AMDT1.py and sent to AMDT2.py and AM1.py

AMDT2.py:

AMDT2.py emulates AMDT2, a digital twin in the protocol. It is given its starting values from the registry enroller and initiates DSTMAKA-1 by authentication with AMDT1.py to produce a shared session key. Afterwards, AMDT2.py starts its own DSTMAKA-2 by authenticating AM2.py to produce a separately distinct session key. AMDT2.py then receives the .gcode and .ngc files from AMDT1.py and sends these files to AM2.py.

AM1.py:

AM1.py emulates AM1, a physical machine in the protocol. It is given its starting values from the registry enroller and initiates DSTMAKA-2 by authentication with AMDT1.py to produce a shared session key. After it has been authenticated it then receives the .gcode and .ngc files from AMDT1.py

AM2.py:

AM2.py emulates AM2, a physical machine in the protocol. It is given its starting values from the registry enroller and initiates DSTMAKA-2 by authentication with AMDT2.py to produce a shared session key. After it has been authenticated it then receives the .gcode and .ngc files from AMDT2.py

Automation.py:

Automation.py serves as a file which automates the process of testing the protocol via running RE.py, AMDT1.py, AMDT2.py, AM1.py and AM2.py in that order. This negates human delay in individually running each file separately. In order to properly run, you must insert the proper file path of each file in RE_path, amdt1_path, amdt2_path, am1_path, and am2_path respectively. When started, Automation.py will initiate each file in a new terminal window and carry out the protocol. In the given GitHub, we have two versions of the Automation.py file. In

the folder denoted as ‘Automated showcase’, the automation.py file uses links to the github to locally run an automated iteration of the program. In the folder denoted as ‘Manual Testing and network changes’, the automated files must be adjusted to possess the corresponding file paths to where you downloaded the needed .py files to run the program. We also have three varieties of Automation.py between these two folders: one for Windows, Mac, and Linux, due to the commands needed to start a terminal window being different given the platform.

Explanation of the Execution cycle of each phase:

Registration Phase:

- AMDT1, AMDT2, AM1, and AM2 bind on port 5050 to the Registry Enroller
- Steps RAD1-RAD3 are initiated for AMDT1, AMDT2, AM1, and AM2 individually
- Once complete all entities disconnect from port 5050 after they have completed their registration

DTSMAKA-1 Phase:

- AMDT1 binds to AMDT2 on socket port 8080
- AMDT1 and AMDT2 execute steps ADT1-ADT4 to authenticate AMDT2
- AMDT1 and AMDT2 execute steps ADT5-ADT7 to authenticate AMDT1

DTSMAKA-2 Phase (AMDT1-AM1):

- AMDT1 binds to AM1 on port 7070
- AMDT1 and AM1execute steps AMA1-AMA4 to authenticate AM1
- AMDT1 and AM1 execute steps AMA5-AMA7 to authenticate AMDT1

DTSMAKA-2 Phase (AMDT2-AM2):

- AMDT2 binds to AM2 on port 9090
- AMDT2 and AM2 execute steps AMA1-AMA4 to authenticate AM2
- AMDT2 and AM2 execute steps AMA5-AMA7 to authenticate AMDT2

File Transfer Phase:

- Maintaining all previous port connections AM1 reads the file path of the .gcode and .ngc file given and sends them to AMDT1
- AMDT1 then receives the files’ data writes its own copy and sends them to AMDT2
- AMDT2 then receives the files’ data writes its own copy and sends them to AM2
- AM2 then receives the files’ data and writes it own copy

Design Decisions:

1) **Elliptic curve encryption of registry enroller values to all entities that connect to it**
    - Elliptic curve encryption was utilized due to the desired need to ensure that the data sent from the registry enroller to any one entity, such as AMDT1, was secure. The values that are being sent over are such things as the entity’s private key, public key, and other forms of ID. The security of this data was vital to keep hidden and the infeasibility of being able to decrypt ECC data without the

proper session key along with its smaller size in comparison to something like RSA made it the most alluring choice.

2) **JSON formatting of sent messages**
    - The protocol demands that numerous messages be sent back and forth during authentication. With JSON, we would be able to alleviate the overhead for data transmission and improve overall performance with its lightweight benefits.

3) **Automation file integration**
    - The integration of an automation file in our protocol was for the use of testing multiple cases of our protocol in rapid succession, thus removing human latency in our test data. Aside from that, the file also allows one to simply run the protocol once via altering the number of iterations of the for loop and overall be a less tedious process of running our protocol rather than opening 5 individual windows to run each python file.

4) **Greenflag integration in Automation file**
    - In the Automation files depending on the platform, they describe a socket connection called greenflag. The point of this connection was to ease the automation process of testing multiple cases of our protocol. When running multiple tests in quick succession between two computers, issues arose. Constantly closing and opening sockets made it difficult to gather concise and accurate data. Therefore, the greenflag socket connection was meant to help communicate between the two computers when each aspect of the protocol was ready to start and connect via socket. An example being that once AMDT1 has finished registering, greenflag sends a message to the other computer to indicate to it that it can run AMDT2. In the python code given in the github, this feature is commented out as to allow the automation python file to run without need of another device running the protocol. It is recommended to use this feature should one want to run a large quantity of tests of the protocol in a cross-platform environment.

5) **OFB encryption**
    - OFB was chosen due to its ability to not need padding when sending messages back and forth. We had previously tried using ECB encryption, but the need to both pad and unpad our message proved to cause several errors to occur where we needed to fit our message into a 16 byte based size. With OFB, the only additional requirement to its encryption and decryption was the use of an iv value. It also encompassed other qualities that supported our work such as being simple to implement and its low overhead when encrypting and decrypting our sent message and or files.

Execution Steps:

Preamble:

The github possesses two folders; Automated showcase and Manual Testing and network changes. Both folders possess the same protocol, but are meant to be used for differing scenarios. The Automated showcase folder is meant to be used in a local environment in which specifically one user runs the protocol using their os-specific automation.py file. The automation file sources the protocol using http links to the github to read and write the .py files and execute the protocol. This serves as an easy and effective way to quickly test and observe the protocol.

The Manual Testing and network changes folder is intended to be used in the case of multiple users wanting to test the protocol in a cross platform environment. In this case one must download each python file of the protocol and alter the ip addresses, and pathfinding to fit the form of network they wish to test. An example is if user 1 wishes to run AMDT1 and AMDT2, while another user is running RE, AM1, and AM2. Thus, the ip addresses must be altered to accommodate this.

Automated showcase execution:

- Download one of the following depending on your os from the Automated showcase folder:
    - Automated_windows_githubsourced.py
    - Automated_Mac_githubsourced.py
    - Automated_Linux_githubsourced.py
- Run the file either in visual studio or on your computer's terminal
- Various terminal windows will appear showcasing each entity in the protocol and its execution

Manual Testing and network changes execution:

- Download the Manual Testing and network changes folder and save it in a safe space
- Go into each python file and where it states ‘INSERT Desired IP ADDRESS OF _ BLANK’ put in the ip address in terms of which devices will act as any one of the entities in the protocol setup ie. one computer acts as AMDT1 while another acts as AMDT2
- The socket programming connections are as follows:
    - All entities temporarily bind to port 5050 with RE
    - AMDT2 binds to AMDT1 on port 8080
    - AM1 binds to AMDT1 on port 7070
    - AM2 binds to AMDT2 on port 9090
- Something to note is that this execution is intended for cross platform testing, as such one computer does not need to access all python files, only the files with which they wish to act as in the protocol

- In the proper automation python based on os, on the variables that list:'FILE PATH OF __ GOES HERE' paste the file path to python files you wish to act as in the protocol
- Omit any code in the automation file that would start a python file you do not wish to serve as the host for ie. if you are acting as AMDT1 and another computer is acting as AMDT2 omit the syntax that starts AMDT2 on your computer
- Run the automation file in either visual studio or on your computer's terminal
- Various terminal windows will appear showcasing each entity in the protocol and its execution