SAND2023-04528R

Active High Assurance Authentication Protocol (AHAAP)

Adrian R. Chavez, Tam D. Lee, Juan A. Martinez, Matthew S. Stroble

May 2023

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

LIST OF FIGURES

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
AHAAP	Active High Assurance Authentication Protocol
CAC	Common Access Card
C&C	Command & Control
DER	Distributed Energy Resource
GUI	Graphical User Interface
OS	Operating System
OTA	Over-the-Air
PKI	Public Key Infrastructure
RDP	Randomized DUF Pattern
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
QEMU	Quick Emulator

1. OVERVIEW

The AHAAP Maturation Project involves maturation and evaluation of a patented zero-trust tamper-resistant high-assurance session-less dynamic and active device authentication protocol that simultaneously authenticates identity and provides integrity verification in a single step, substantially reducing the risk of cyberattack, and eliminating the need for costly and complex conventional communication security systems requirements (i.e., cryptography, Public Key Infrastructure (PKI), and key management).

These cybersecurity attributes of the technology must be preserved when applying the technology to different cybersecurity solutions, including Command & Control (C&C), Over-the-Air (OTA) update, Common Access Card (CAC), and distributed energy resource (DER) implementations, among others.

The technology research objective is to test and verify that the cybersecurity attributes of the technology are not degraded in different cybersecurity applications. The primary technology development objective is to build minimum viable products to demonstrate the technology addresses today's cybersecurity threats so that prospective investors, strategic partners, regulatory agencies, and commercial customers can interact with and assess the protection assured by the technology. The AHAAP Maturation Project goal is to develop, test, and validate one or more AHAAP implementations.

The AHAAP Maturation Project tasks are: (i) engineer AHAAP implementation software, (ii) build a functional prototype that implements the AHAAP software for demonstration, testing, analysis, and evaluation purposes, and (iii) generate a report detailing the results of the AHAAP C&C software and hardware implementation.

The final project deliverables are: (i) AHAAP software implementation and prototype, (ii) a report from Sandia National Laboratories detailing the results of the AHAAP implementations.

2. DIGITALLY UNCLONABLE FUNCTIONS PROTOCOL

The DUF protocol is a multi-party protocol to validate the integrity of data shared between two parties. The protocol depends on a secret key that is stored within a tamper resistant device where an XOR and a SHA512 hash can be performed (the red cylinder in Figure 1). The secret data stored is called a Randomized DUF Pattern (RDP) and is generated by the user, independent of the system manufacturer. The RDP can be based off a user's handwritten signature or a pin and can be derived in conjunction with a true source of entropy. Both the manufacturer and the end device owner would store a copy of the RDP. In Figure 1, the left side of the diagram represents a vehicle manufacturer that may have occasional firmware updates. The right side of the diagram represents the vehicle that will receive and validate the firmware update through the DUF protocol. The manufacturer would process the firmware combined with meta-data (such as time, version, and location information) through the DUF tamper-resistant chip to produce a DUF signature. The firmware and meta-data would be communicated, in cleartext, along with the DUF signature to the vehicle. The vehicle would then perform the same computation using its DUF chip with the secret key to validate the integrity of the firmware update. If the signatures produced by the vehicle and manufacturer match, then the firmware update is applied. Otherwise, the firmware update is rejected. The DUF protocol operates in this fashion and can be applied to a broad range of applications outside of electric vehicles. The DUF protocol is described in more detail in the sections that follow.

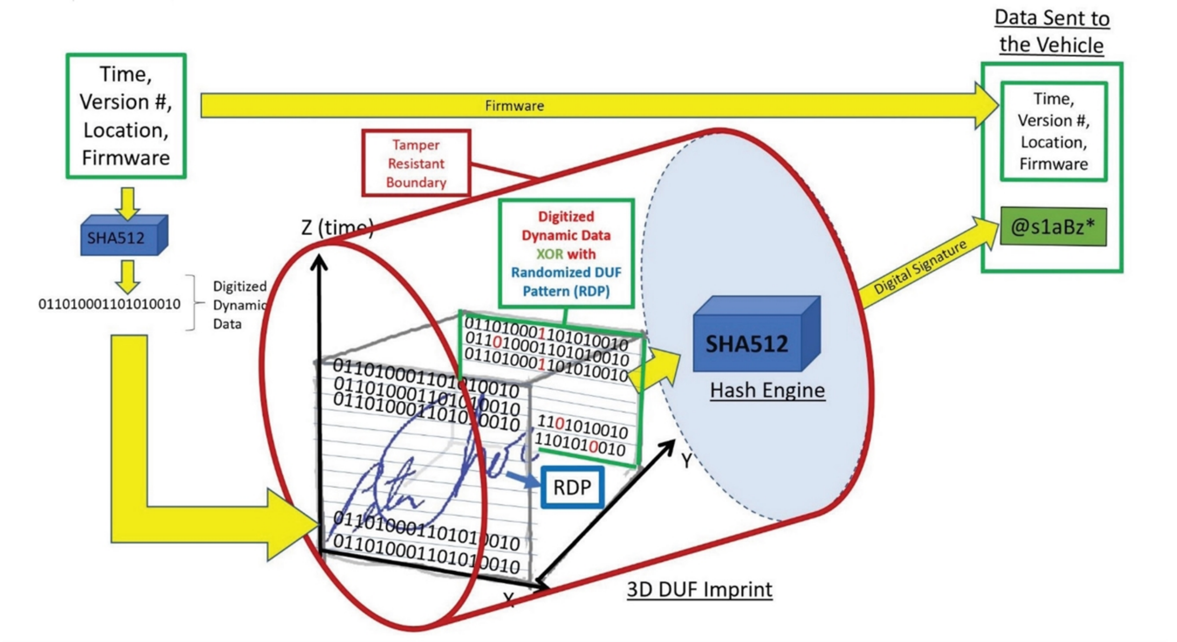

Data Sent to the Vehicle
Time, Version #, Location, Firmware
Firmware
Time, Version #, Location, Firmware
Tamper Resistant Boundary
SHA512
Z (time)
Digitized Dynamic Data XOR with Randomized DUF Pattern (RDP)
@s1aBz*
Digital Signature
011010001101010010
Digitized Dynamic Data
SHA512
Hash Engine
RDP
Y
X
3D DUF Imprint

Figure 1: The DUF protocol between two parties for a Secure Firmware-Over-the-Air (S-FOTA) application.

2.1. DUF Overview

The DUF protocol consists of two stages:

1. Commissioning
2. Operating

The Commissioning stage creates a randomized Digitally Unclonable Function (DUF) Pattern (RDP) that is stored on both the server and client systems. This stage only happens once at initialization. Once the commissioning phase is complete, the next stage is the Operating stage which executes the DUF protocol.

2.2. DUF Commissioning

For this implementation, the RDP is created by randomly generating 512-bit strings XOR'd together continuously until the user presses a key. Once the user presses a key, the RDP is written to a file named duf.txt by default. The content of the duf.txt file is then sent to the client system to also write to the same file name by default. The communications are TLS encrypted using digital certificates. The certificates are self-signed certificates for this implementation but can be substituted in for certificates signed by a certificate authority.

2.3. DUF Operation

Once the RDP has been stored on both the client and server, the DUF protocol can be enabled and advanced to the operating phase. The client system starts by listening for data to be received from the client (such as a firmware update). Once the data is received by the client, the server will also send a DUF signature of the data that will be validated using the duf.txt secret key that was communicated during the commissioning phase of the protocol. The data also includes meta-data, such as timestamp, version, and location information at the header. The client processes all data (including meta-data) through the DUF protocol to confirm that the DUF signature matches the signature communicated from the server. The communications are also TLS encrypted using the same certificates as the commissioning stage. If the signatures match, then the data is trusted, otherwise the data is not trusted. The use case for this implementation is a software update being provided by the server (a vehicle manufacturer) to the client (an electric vehicle).

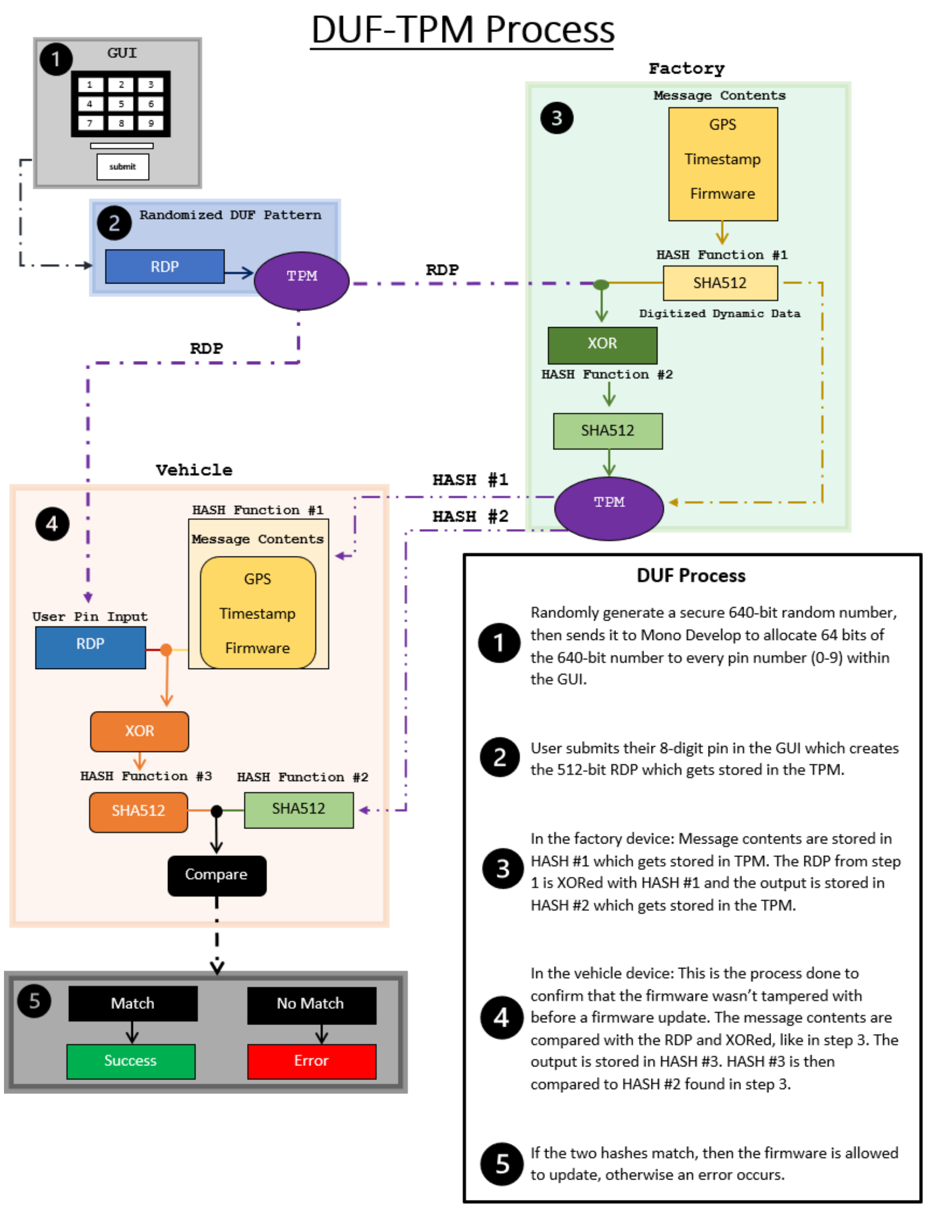
3. DIGITALLY UNCLONABLE FUNCTIONS IMPLEMENTATION DETAILS

3.1. DUF-TPM

The initial idea was to use the TPM chip to create and store the Random DUF Pattern (RDP). The TPM chip would randomly generate a secure 80-byte (640-bit) number into an array. Each set of 8 bytes (64 bits) of the 80-byte number would represent a pin digit value (0-9). The user would type their own unique 8-digit pin into the user interface and submit it to the TPM chip. The user's pin would determine which set of 8-bytes from the 80-byte number were used and the order they would be represented in. This resulting number makes up the 64-byte (512-bit) RDP. The RDP would then be sent from the TPM chip to the factory and vehicle devices. In the factory device, the message contents (containing the GPS, timestamp, and firmware) are hashed which creates HASH 1. The RDP is then XOR-ed with HASH 1 and the result creates HASH 2. HASH 2 and HASH 1 are stored in the TPM chip for safe keeping. In the vehicle device, HASH 1 and the RDP are sent over from the TPM chip and are XOR-ed together. The result of the XOR creates HASH 3. HASH 2 is then sent over from the TPM chip and compared with HASH 3. If the two hashes match, then there is a successful update otherwise an error is recorded. The DUF-TPM process is shown in Figure 2, however there were certain security concerns surrounding this process with the TPM chip.

3.1.1. DUF-TPM ISSUES

The TPM chip is a dedicated microcontroller that has numerous components that could be used to implement DUF securely. Most notably, the TPM chip has a built-in random number generator, persistent memory for key storage, and a SHA-1 hash generator. However, TPM chips are very limited as they are fixed-function devices. TPMs are also primarily used for key storage, however, they have a limited amount of persistent memory to store keys. This can be a problem because if a user wanted to encrypt something with a key, they would need to take the key out of the TPM chip and release back into memory which may have been compromised during this process. This became an increased security risk throughout our analysis which prompted us to seek out more secure avenues to execute the DUF protocol.



DUF Process

1 Randomly generate a secure 640-bit random number, then sends it to Mono Develop to allocate 64 bits of the 640-bit number to every pin number (0-9) within the GUI.

2 User submits their 8-digit pin in the GUI which creates the 512-bit RDP which gets stored in the TPM.

3 In the factory device: Message contents are stored in HASH #1 which gets stored in TPM. The RDP from step 1 is XORed with HASH #1 and the output is stored in HASH #2 which gets stored in the TPM.

4 In the vehicle device: This is the process done to confirm that the firmware wasn't tampered with before a firmware update. The message contents are compared with the RDP and XORed, like in step 3. The output is stored in HASH #3. HASH #3 is then compared to HASH #2 found in step 3.

5 If the two hashes match, then the firmware is allowed to update, otherwise an error occurs.

Figure 2: The DUF process and flowchart of the algorithm

3.1.2. QEMU OP-TEE Branch

The OP-TEE branch contains multiple repositories that were designed to implement the complete DUF protocol in a user intuitive process that included a Graphical User Interface (GUI) for DUF RDP PIN registration. The OP-TEE branch utilizes several technologies to aid in developing with ARM Trusted Zone technology like OP-TEE Operating System (OS) and MonoDevelop. OP-TEE OS is the dedicated OS that runs inside ARM Trusted Zone and the OP-TEE driver that allows for interaction between the Untrusted Execution Environment and the Trusted Execution Environment. For simplicity these two technologies will be referred to as OP-TEE. OP-TEE is currently developed by a company named Linaro and is overseen by the Trusted Firmware Foundation. The OP-TEE framework was specifically developed using the Ubuntu Linux Operating system. During the development for the OP-TEE specific implementation of the DUF protocol OP-TEE proved to be quite challenging to develop a proper implementation. Due to a number of challenges detailed in later sections, a successful implementation of the complete DUF Protocol was not possible, only a partial implementation was able to be achieved. The current implementation includes the DUF server running inside of OP-TEE in the ARM Trusted Zone environment and a dedicated GUI application that is used for the registration of a unique RDP PIN sent from the GUI and stored inside of the ARM Trusted Zone. The GUI application runs in the Untrusted Execution Environment also referred to as Normal World. The GUI can be run from another computer on the same network.

3.1.2.1. Virtual Machine

This project requires VMware Workstation Pro v16 or greater if on Windows and VMware fusion pro 12 or greater if on mac. The Development Environment must be based on Ubuntu 22.04, 200GB of storage or higher, 16GB of ram and 8 CPU processor cores.

3.1.2.2. OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) designed to be a companion to a non-secure Linux kernel running on ARM Cortex-A cores using Trusted Zone technology. OP-TEE supports both ARMv7 and ARMv8 architectures in both 32-bit and 64-bit. This project utilizes the ARMv7 32-bit architecture to implement the Quick Emulator (QEMU) version and is supported by the Rock Pi 4B v1.5 board.

3.1.2.2.1. OP-TEE Installation

Run the following command from the Linux terminal to install all the required packages for Ubuntu 22.04.

```
sudo apt install \
  adb \
  acpica-tools \
  autoconf \
  automake \
  bc \
  bison \
  build-essential \
  ccache \
  cscope \
  curl \
  device-tree-compiler \
  e2tools \
  expect \
  fastboot \
  flex \
  ftp-upload \
  gdisk \
  libattr1-dev \
  libcap-dev \
  libfdt-dev \
  libftdi-dev \
  libglib2.0-dev \
  libgmp3-dev \
  libhidapi-dev \
  libmpc-dev \
  libncurses5-dev \
  libpixman-1-dev \
  libslirp-dev \
  libssl-dev \
  libtool \
  libusb-1.0-0-dev \
  make \
  mtools \
  netcat \
  ninja-build \
  python3-cryptography \
  python3-pip \
  python3-pyelftools \
  python3-serial \
  python-is-python3 \
  rsync \
  swig \
  unzip \
  uuid-dev \
  xdg-utils \
  xterm \
  xz-utils \
  zlib1g-dev \
  repo
```

Clone the OPTEE repo to the user's home directory in Linux. Change the terminal location to the newly created directory that was created during the cloning process, OPTEE. From this root folder run the following commands one at a time until they complete. You may get configuration questions asked about setting up google repo during the first command, just respond with yes.

```
repo init -u https://github.com/OP-TEE/manifest.git
repo sync
cd build
```

From the root of the repo, change directories to the build folder and run the following command.

```
make toolchains
```

This will build the tool chains required for the specific QEMU installation ARMv7 32-bit.
If your intention is to do debugging within OPTEE, ASLR must be disabled by setting the flag CFG_CORE_ASLR=n in the Linux terminal. This can be done with the following command from the Linux Terminal.

```
export CFG_CORE_ASLR=n
```

At this point, a complete compile of OPTEE is ready with the DUF server code. Run the following command to start the lengthy compile.

```
make run
```

Once the compiling of OPTEE-OS finishes, the system will automatically start the QEMU environment. Two additional terminal tabs labeled "Secret World" and "Normal World" will appear.

3.1.2.2.2. DUF-GUI

The DUF-GUI is a program that allows the user to input the unique RDP pin number to be stored into the Arm Trusted Zone environment. It was developed using MonoDevelop and C# so that it can run on any computing device that is supported, Windows, Linux, and Mac. This interface is what the manufacture would use to install the secret code on to the target device running the DUF server that must be verified in the DUF Protocol.

3.1.2.2.2.1. DUF-GUI Installation

Run the following lines from the Linux terminal. This will install the latest version of MonoDevelop onto the Development system.

```
sudo apt install apt-transport-https dirmngr

sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF

echo "deb https://download.mono-project.com/repo/ubuntu vs-bionic main" | sudo tee
/etc/apt/sources.list.d/mono-official-vs.list

sudo apt update

sudo apt install monodevelop
```

MonoDevelop should be successfully installed at this point.
Open the DUF-GUI project repo in MonoDevelop and press the play button to start the program.

3.1.2.2.3. Running OP-TEE Branch

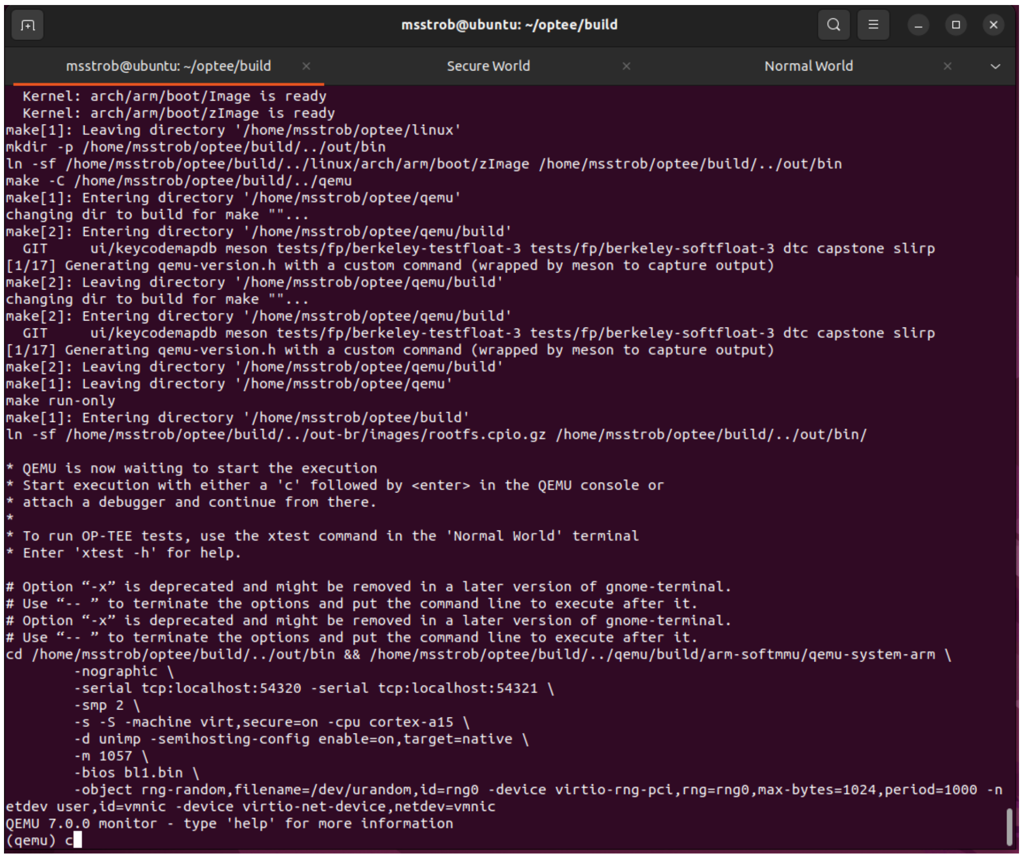
From the OPTEE repo folder, change to the builds folder and run the following command.

```
make -j8 run
```

Once the system completes the task you should see two new terminal tabs in the terminal window, Normal World and Secure World. Once all the text lines have stopped scrolling in the Normal World terminal, press the following key for the terminal prompt in the original terminal that you run the previous command.

```
c
```

You will not see anything happen in the initial terminal, but you will see stuff happening in the Secure World and Normal World terminals.



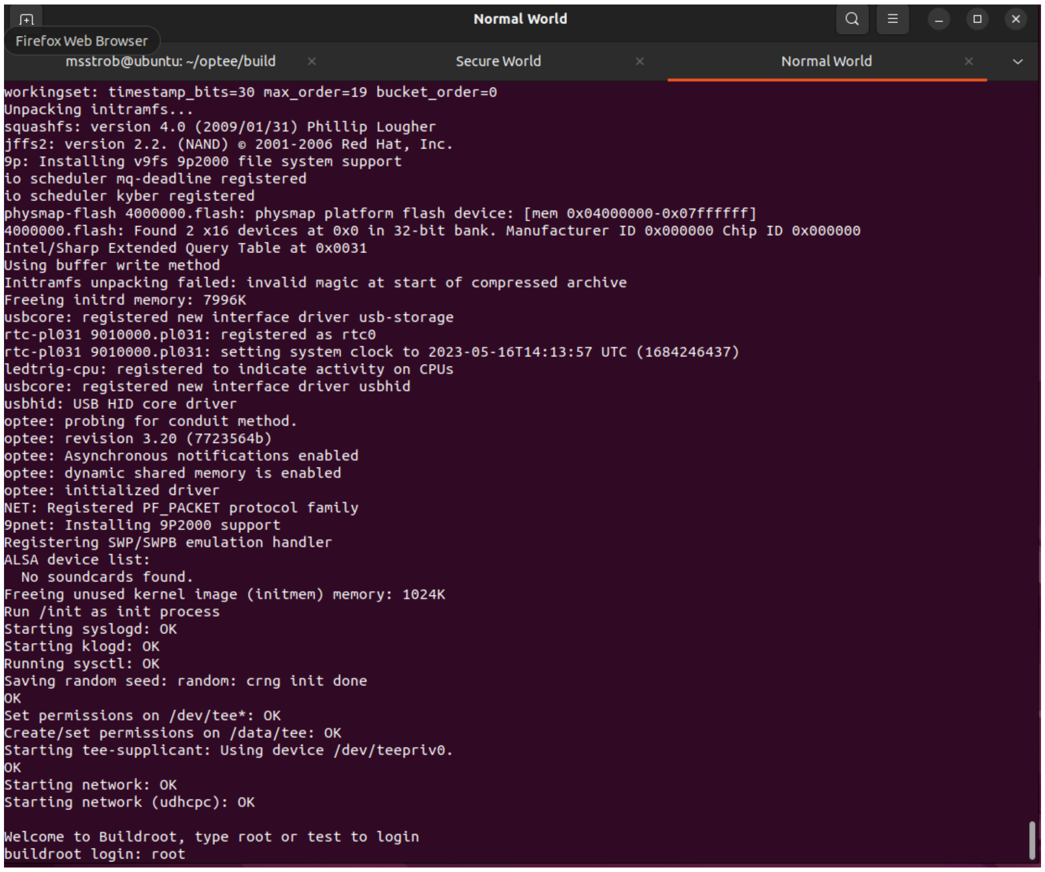
```
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/zImage is ready
make[1]: Leaving directory '/home/msstrob/optee/linux'
mkdir -p /home/msstrob/optee/build/../out/bin
ln -sf /home/msstrob/optee/build/../linux/arch/arm/boot/zImage /home/msstrob/optee/build/../out/bin
make -C /home/msstrob/optee/build/../qemu
make[1]: Entering directory '/home/msstrob/optee/qemu'
changing dir to build for make ""...
make[2]: Entering directory '/home/msstrob/optee/qemu/build'
  GIT     ui/keycodemapdb meson tests/fp/berkeley-testfloat-3 tests/fp/berkeley-softfloat-3 dtc capstone slirp
[1/17] Generating qemu-version.h with a custom command (wrapped by meson to capture output)
make[2]: Leaving directory '/home/msstrob/optee/qemu/build'
changing dir to build for make ""...
make[2]: Entering directory '/home/msstrob/optee/qemu/build'
  GIT     ui/keycodemapdb meson tests/fp/berkeley-testfloat-3 tests/fp/berkeley-softfloat-3 dtc capstone slirp
[1/17] Generating qemu-version.h with a custom command (wrapped by meson to capture output)
make[2]: Leaving directory '/home/msstrob/optee/qemu/build'
make[1]: Leaving directory '/home/msstrob/optee/qemu'
make run-only
make[1]: Entering directory '/home/msstrob/optee/build'
ln -sf /home/msstrob/optee/build/../out-br/images/rootfs.cpio.gz /home/msstrob/optee/build/../out/bin/

* QEMU is now waiting to start the execution
* Start execution with either a 'c' followed by <enter> in the QEMU console or
* attach a debugger and continue from there.
*
* To run OP-TEE tests, use the xtest command in the 'Normal World' terminal
* Enter 'xtest -h' for help.

# Option “-x” is deprecated and might be removed in a later version of gnome-terminal.
# Use “-- ” to terminate the options and put the command line to execute after it.
# Option “-x” is deprecated and might be removed in a later version of gnome-terminal.
# Use “-- ” to terminate the options and put the command line to execute after it.
cd /home/msstrob/optee/build/../out/bin && /home/msstrob/optee/build/../qemu/build/arm-softmmu/qemu-system-arm \
        -nographic \
        -serial tcp:localhost:54320 -serial tcp:localhost:54321 \
        -smp 2 \
        -s -S -machine virt,secure=on -cpu cortex-a15 \
        -d unimp -semihosting-config enable=on,target=native \
        -m 1057 \
        -bios bl1.bin \
        -object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-pci,rng=rng0,max-bytes=1024,period=1000 -n
etdev user,id=vmnic -device virtio-net-device,netdev=vmnic
QEMU 7.0.0 monitor - type 'help' for more information
(qemu) c
```

Figure 3: Initialization of QEMU Environment

There will cause the system to initialize and quite a few lines will be show in the terminal windows. Once the process is complete the Normal World will ask for the buildroot login.
Type in the username:

```
root
```

Press the enter key.

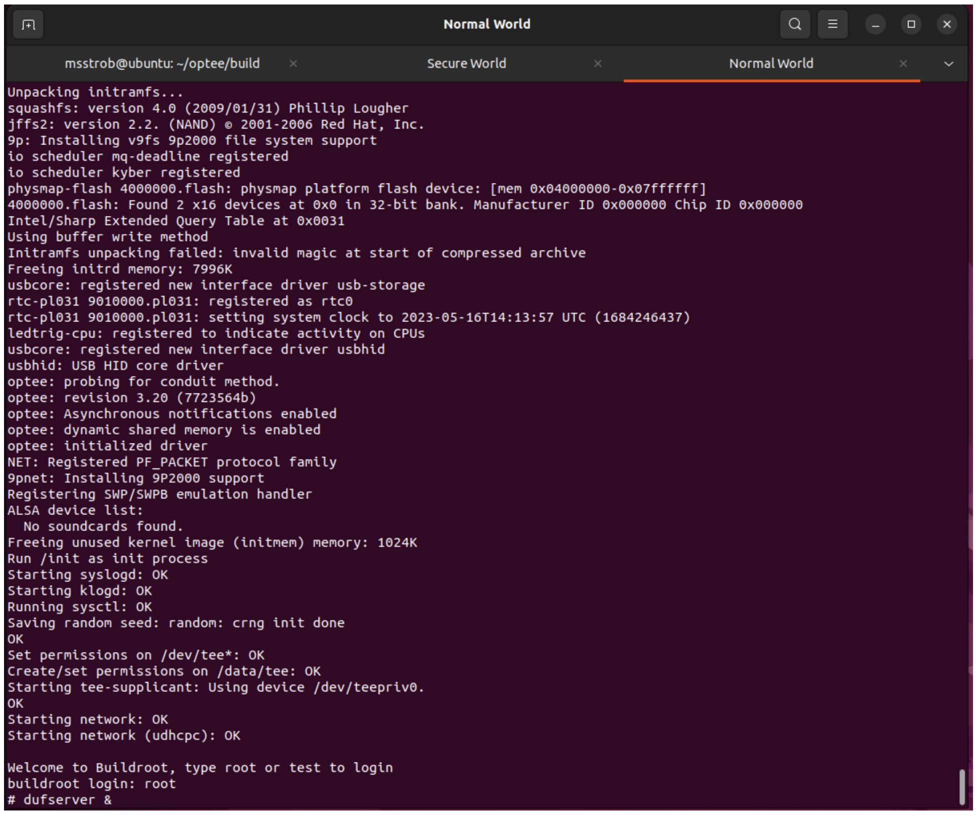


```
workingset: timestamp_bits=30 max_order=19 bucket_order=0
Unpacking initramfs...
squashfs: version 4.0 (2009/01/31) Phillip Lougher
jffs2: version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
9p: Installing v9fs 9p2000 file system support
io scheduler mq-deadline registered
io scheduler kyber registered
physmap-flash 4000000.flash: physmap platform flash device: [mem 0x04000000-0x07ffffff]
4000000.flash: Found 2 x16 devices at 0x0 in 32-bit bank. Manufacturer ID 0x000000 Chip ID 0x000000
Intel/Sharp Extended Query Table at 0x0031
Using buffer write method
Initramfs unpacking failed: invalid magic at start of compressed archive
Freeing initrd memory: 7996K
usbcore: registered new interface driver usb-storage
rtc-pl031 9010000.pl031: registered as rtc0
rtc-pl031 9010000.pl031: setting system clock to 2023-05-16T14:13:57 UTC (1684246437)
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
optee: probing for conduit method.
optee: revision 3.20 (7723564b)
optee: Asynchronous notifications enabled
optee: dynamic shared memory is enabled
optee: initialized driver
NET: Registered PF_PACKET protocol family
9pnet: Installing 9P2000 support
Registering SWP/SWPB emulation handler
ALSA device list:
  No soundcards found.
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: random: crng init done
OK
Set permissions on /dev/tee*: OK
Create/set permissions on /data/tee: OK
Starting tee-supplicant: Using device /dev/teepriv0.
OK
Starting network: OK
Starting network (udhcpc): OK

Welcome to Buildroot, type root or test to login
buildroot login: root
```

Figure 4: Root Login into QEMU Normal World

If the login is successful, you will see the message in the Normal World Terminal "Made it here!". This will allow the user to begin execution of programs in the QEMU environment. Run the following command in the Normal World terminal.

```
dufserver &
```



```
Unpacking initramfs...
squashfs: version 4.0 (2009/01/31) Phillip Lougher
jffs2: version 2.2. (NAND) © 2001-2006 Red Hat, Inc.
9p: Installing v9fs 9p2000 file system support
io scheduler mq-deadline registered
io scheduler kyber registered
physmap-flash 4000000.flash: physmap platform flash device: [mem 0x04000000-0x07ffffff]
4000000.flash: Found 2 x16 devices at 0x0 in 32-bit bank. Manufacturer ID 0x000000 Chip ID 0x000000
Intel/Sharp Extended Query Table at 0x0031
Using buffer write method
Initramfs unpacking failed: invalid magic at start of compressed archive
Freeing initrd memory: 7996K
usbcore: registered new interface driver usb-storage
rtc-pl031 9010000.pl031: registered as rtc0
rtc-pl031 9010000.pl031: setting system clock to 2023-05-16T14:13:57 UTC (1684246437)
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
optee: probing for conduit method.
optee: revision 3.20 (7723564b)
optee: Asynchronous notifications enabled
optee: dynamic shared memory is enabled
optee: initialized driver
NET: Registered PF_PACKET protocol family
9pnet: Installing 9P2000 support
Registering SWP/SWPB emulation handler
ALSA device list:
  No soundcards found.
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: random: crng init done
OK
Set permissions on /dev/tee*: OK
Create/set permissions on /data/tee: OK
Starting tee-supplicant: Using device /dev/teepriv0.
OK
Starting network: OK
Starting network (udhcpc): OK

Welcome to Buildroot, type root or test to login
buildroot login: root
# dufserver &
```

Figure 5: Launching Dufserver

This will start the dufserver running in QEMU in the Normal World. Note the terminating input indicator will not appear anymore after this point. At this point you need to open the Ubuntu File Browser and navigate to the cloned repo DUF-GUI. Right Click the file "DUF_TrustedZone.sln" and choose "Open With other Application" then choose "View All Applications", select MonoDevelop and Select at the top to open the project folder in the MonoDevelop IDE. Click the big play button to launch the program. Once you see the interface, input your secret 8 digit PIN number and press the "submit" button.

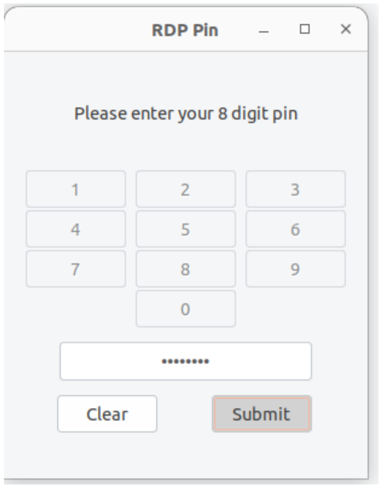
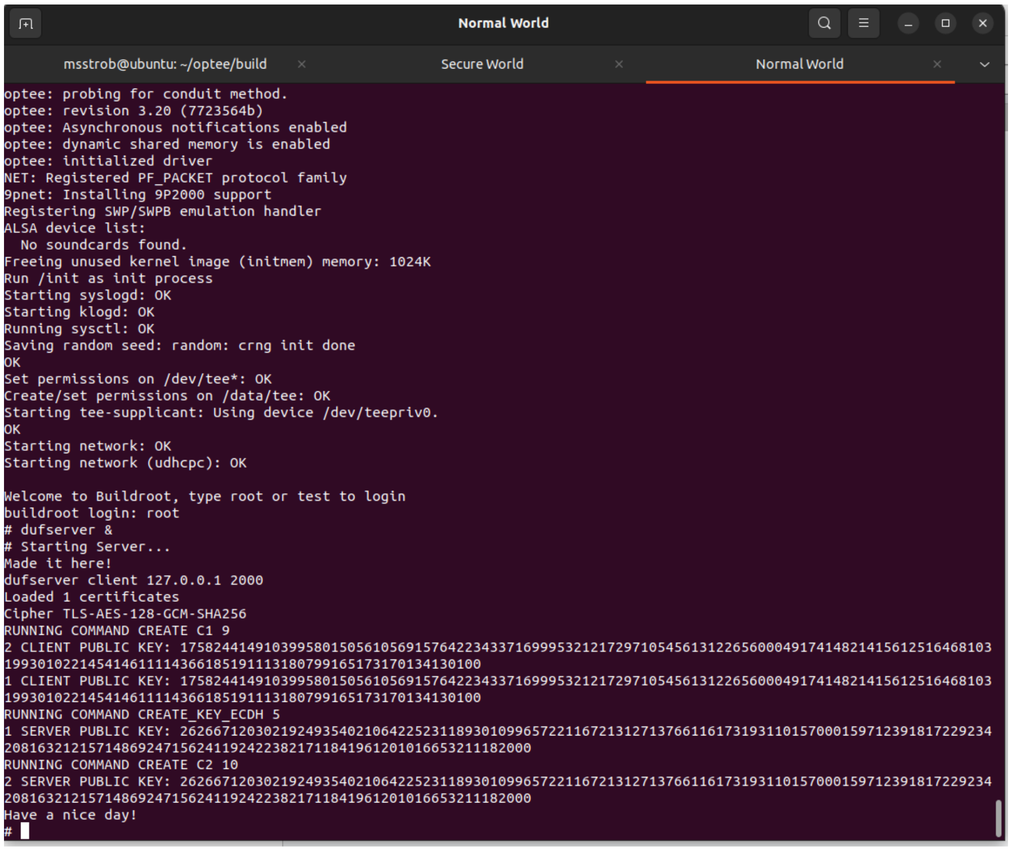


Figure 6: DUF-GUI RDP Pin Input

It is time to go back to the Normal World Linux Terminal window and run the following command. **Note:** if you see a popup window asking if you want to wait or close the DUF-GUI program before you have run the command below, click wait before running the following command.

```
dufserver client 127.0.0.1 2000
```



```
optee: probing for conduit method.
optee: revision 3.20 (7723564b)
optee: Asynchronous notifications enabled
optee: dynamic shared memory is enabled
optee: initialized driver
NET: Registered PF_PACKET protocol family
9pnet: Installing 9P2000 support
Registering SWP/SWPB emulation handler
ALSA device list:
  No soundcards found.
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: random: crng init done
OK
Set permissions on /dev/tee*: OK
Create/set permissions on /data/tee: OK
Starting tee-supplicant: Using device /dev/teepriv0.
OK
Starting network: OK
Starting network (udhcpc): OK

Welcome to Buildroot, type root or test to login
buildroot login: root
# dufserver &
# Starting Server...
Made it here!
dufserver client 127.0.0.1 2000
Loaded 1 certificates
Cipher TLS-AES-128-GCM-SHA256
RUNNING COMMAND CREATE C1 9
2 CLIENT PUBLIC KEY: 17582441491039958015056105691576422343371699953212172971054561312265600049174148214156125164681031
99301022145414611114366185191113180799165173170134130100
1 CLIENT PUBLIC KEY: 17582441491039958015056105691576422343371699953212172971054561312265600049174148214156125164681031
99301022145414611114366185191113180799165173170134130100
RUNNING COMMAND CREATE_KEY_ECDH 5
1 SERVER PUBLIC KEY: 26266712030219249354021064225231189301099657221167213127137661161731931101570001597123918172292342
0816321215714869247156241192422382171184196120101665321118200
RUNNING COMMAND CREATE C2 10
2 SERVER PUBLIC KEY: 26266712030219249354021064225231189301099657221167213127137661161731931101570001597123918172292342
0816321215714869247156241192422382171184196120101665321118200
Have a nice day!
#
```

Figure 7: Execution of DUF Server in local environment and successful register of RDP Pin and HASH checking

If the communication is successful, the Normal World will show an end message “Have a nice day!” and in the Secure World Terminal you will see the message “Contents Verified”. This means that the RDP was successfully generated, and the resulting HASH comparisons match up. The Secure World Terminal will show the RDP setup tables for verification.

3.1.2.2.4. OP-TEE Branch Issues

The main framework that is used to work with ARM Trusted Zone technology is called OP-TEE. This framework makes it possible to develop secure applications that work with ARM Trusted Zone Technologies. When the DUF Project first started development of the QEMU implementation of the DUF Protocol, the OP-TEE project was run by Linaros Community Development Division. Over the life of this project Linaro donated the OP-TEE project to the Trusted Firmware Foundations where developers completely rewrote the whole project. As new developers were brought onto the DUF Project, it was discovered that it became technically infeasible to set up new development environments for additional personnel and get the current project code operational in new development environments. Extensive debugging and research were required to track down the issues as the new owners of OP-TEE did not provide any change logs to the project. The following issues were discovered during this extensive debugging and upgrading of the DUF Project code from the old OP-TEE project to the new fully rewritten OP-TEE project released by the Trusted Firmware Foundation. Extensive debugging was required to bring the project up to date and in line with all the undocumented changes to the rewritten OP-TEE code base that is available today.

- OP-TEE project complete rewrite
- Required Ubuntu changed 18.04 to 22.04
- Old Package Requirements removed from APT Package Manager
- New Package Requirements
- Original OP-TEE source code was removed from GitHub

3.1.2.2.4.1. The OP-TEE Project

Throughout the development of the DUF Projects OP-TEE Branch, the OP-TEE project changed hands from the original developer Linaro and its Community Division when they donated the project to the Trusted Firmware Foundation. Trusted Firmware Foundation took ownership of the OP-TEE project, and the projects code base was completely rewritten. The original source code written by Linaro was removed from GitHub upon the first official release by the Trusted Firmware Foundation. Trusted Firmware Foundations OP-TEE rewrite changed many of the package requirements from the DUF Projects initial version of OP-TEE. The OP-TEE rewrite caused complications as the change in packages requirements was not documented, only the message “Older version of OP-TEE are no longer supported”.

The new version of OP-TEE required Ubuntu 22.04 over the original Ubuntu 18.04. Ordinarily this is not an issue in Linux development, but the packages in the original OP-TEE requirements are no longer on APT package manager, nor are the source codes for those packages available on GitHub. The way forward was to update everything to the latest version of OP-TEE. The complete rewrite caused unforeseen complications that took extensive investigation to correct errors caused by the Trusted Firmware Foundations rewrite of OP-TEE Project.

3.1.2.2.4.2. RockPi 4B v1.5

A hardware implementation of the OP-TEE branch was targeted for the RockPi 4B v1.5 single board computer. The official OP-TEE project documentation specifically lists this hardware as being compatible with OP-TEE. There is zero documentation on the official OP-TEE documentation on how to run OP-TEE on physical hardware for this hardware even though their site explicitly states support for this piece of hardware. There is an installer to install OP-TEE onto the hardware but there is no way to verify that OP-TEE is even running on the device or instructions on how to launch an application that is compiled and installed with OP-TEE in the Trusted Execution Environment. A special UART cable was used to gain terminal access through the GPIO ports as listed on the hardware manufactures website, but the hardware provided zero responses or output to attempts. The GPIO pins did not provide a signal that could be detected using a multi-meter.

3.1.2.2.5. ARM TrustZone

ARM TrustZone architecture provides a security framework that separates execution into two environments, the Normal World and the Secure World. The Secure World is where the user would run any sensitive programs that they would want to protect from outside processes. Figure shows the Secure and non-secure (Normal) worlds architecture for ARM TrustedZones.

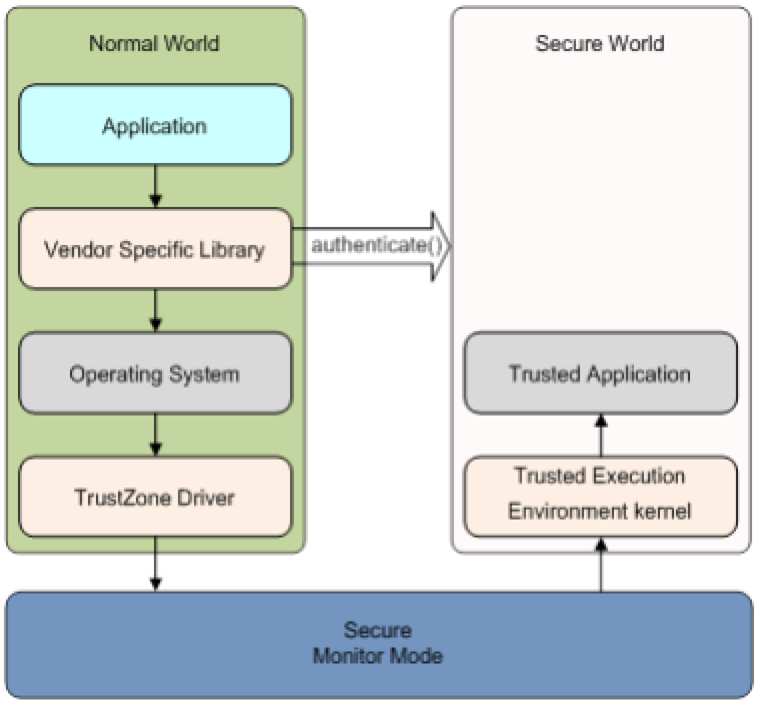


Figure 8: ARM TrustedZone architecture

These two execution states exist on every core of the processor. For example, the user could have an operating system, such as a Linux OS, which would run in the Normal World execution. In the Secure World, the user could have another kernel, which would run trusted applications in the Secure World.

3.2. DUF Physical Environment

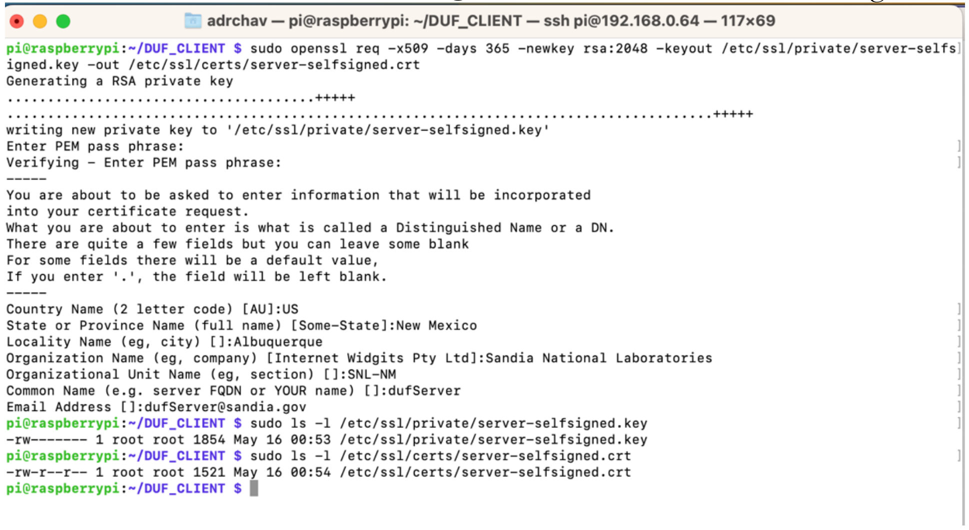
The physical environment consists of two single board computers. The first single board computer is a Rock Pi with ARM TrustZone support running Ubuntu 20 Server OS. The Rock Pi is acting as the DUF server for our implementation. ARM TrustZone was not used for this implementation, but the software is written such that the ARM TrustZone functions can be extracted and placed into the ARM TrustZone Secure World. The second single board computer is a raspberry pi running Raspberry Pi OS Lite. The Raspberry Pi is used to act as the DUF client for this implementation.

3.2.1. Rock Pi Setup

On the **server** system (Ubuntu 22.04.1 for these instructions but Windows should also work):

1. Configure a windows or Linux system with Python3 and openssl `sudo apt-get install python3 net-tools`
2. Configure the IP address to the same subnet as the server. For this demonstration, the ip address is configured to 10.36.1.50/24 `sudo ifconfig eth0 10.36.1.50/24`

3. Generate the certificates and answer all questions that are prompted and set a password on the private key `sudo openssl req -x509 -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/server-selfsigned.key -out /etc/ssl/certs/server-selfsigned.crt` as shown below in Figure :



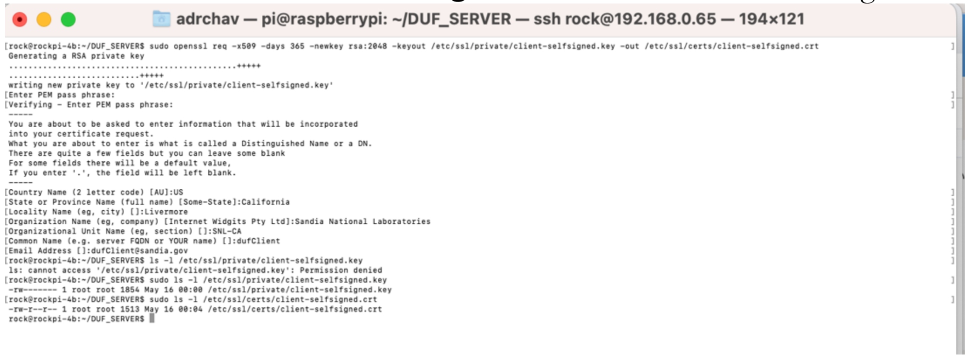
```
pi@raspberrypi:~/DUF_CLIENT $ sudo openssl req -x509 -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/server-selfs
igned.key -out /etc/ssl/certs/server-selfsigned.crt
Generating a RSA private key
....................................+++++
.....................................................................................+++++
writing new private key to '/etc/ssl/private/server-selfsigned.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:New Mexico
Locality Name (eg, city) []:Albuquerque
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Sandia National Laboratories
Organizational Unit Name (eg, section) []:SNL-NM
Common Name (e.g. server FQDN or YOUR name) []:dufServer
Email Address []:dufServer@sandia.gov
pi@raspberrypi:~/DUF_CLIENT $ sudo ls -l /etc/ssl/private/server-selfsigned.key
-rw------- 1 root root 1854 May 16 00:53 /etc/ssl/private/server-selfsigned.key
pi@raspberrypi:~/DUF_CLIENT $ sudo ls -l /etc/ssl/certs/server-selfsigned.crt
-rw-r--r-- 1 root root 1521 May 16 00:54 /etc/ssl/certs/server-selfsigned.crt
pi@raspberrypi:~/DUF_CLIENT $ 
```

Figure 9: The DUF client key generation step to support encrypted communications using TLS

3.2.2. Raspberry Pi Setup

On the **client** system (Ubuntu 22.04.1 for these instructions but Windows should also work):

1. Configure a windows or Linux system with Python3 and openssl `sudo apt-get install python3 net-tools`
2. Configure the IP address to the same subnet as the server. For this demonstration, the ip address is configured to 10.36.1.193/24 `sudo ifconfig eth0 10.36.1.193/24`
3. Generate the certificates and answer all questions that are prompted and set a password on the private key `sudo openssl req -x509 -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/client-selfsigned.key -out /etc/ssl/certs/client-selfsigned.crt` as shown below in Figure :



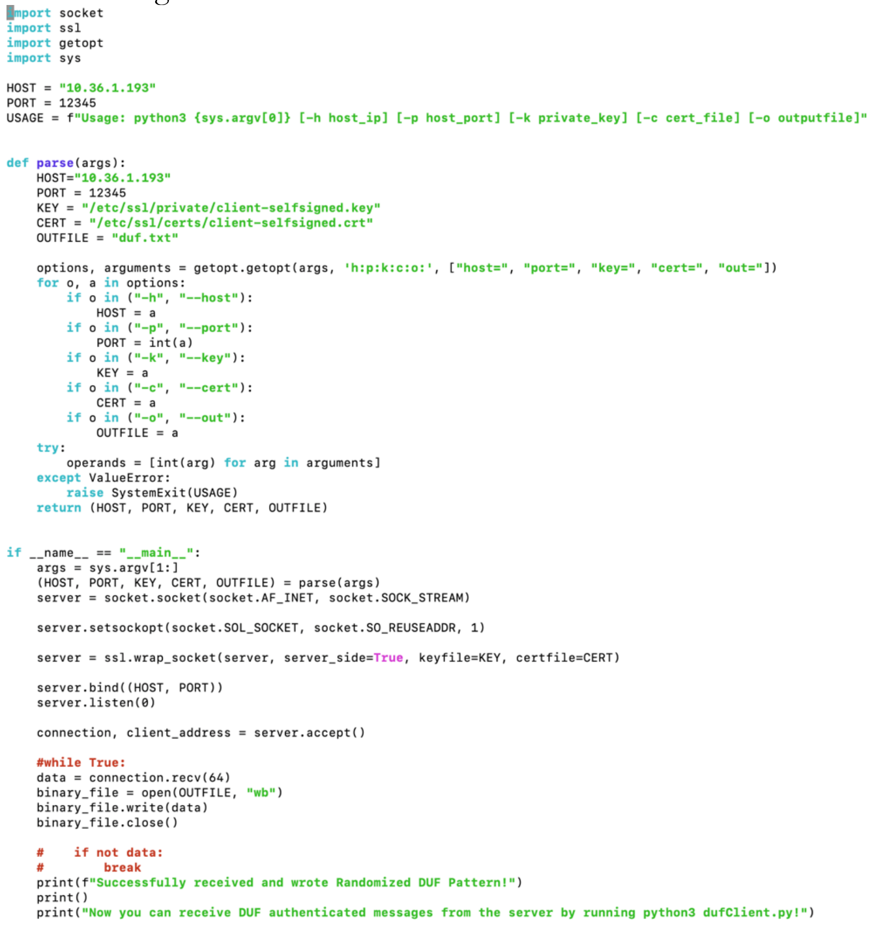
```
[rock@rockpi-4b:~/DUF_SERVER$ sudo openssl req -x509 -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/client-selfsigned.key -out /etc/ssl/certs/client-selfsigned.crt
 Generating a RSA private key
 ...............................................+++++
 ...........................+++++
 writing new private key to '/etc/ssl/private/client-selfsigned.key'
[Enter PEM pass phrase:
[Verifying - Enter PEM pass phrase:
 -----
 You are about to be asked to enter information that will be incorporated
 into your certificate request.
 What you are about to enter is what is called a Distinguished Name or a DN.
 There are quite a few fields but you can leave some blank
 For some fields there will be a default value,
 If you enter '.', the field will be left blank.
 -----
[Country Name (2 letter code) [AU]:US
[State or Province Name (full name) [Some-State]:California
[Locality Name (eg, city) []:Livermore
[Organization Name (eg, company) [Internet Widgits Pty Ltd]:Sandia National Laboratories
[Organizational Unit Name (eg, section) []:SNL-CA
[Common Name (e.g. server FQDN or YOUR name) []:dufClient
[Email Address []:dufClient@sandia.gov
[rock@rockpi-4b:~/DUF_SERVER$ ls -l /etc/ssl/private/client-selfsigned.key
 ls: cannot access '/etc/ssl/private/client-selfsigned.key': Permission denied
[rock@rockpi-4b:~/DUF_SERVER$ sudo ls -l /etc/ssl/private/client-selfsigned.key
 -rw------- 1 root root 1854 May 16 00:00 /etc/ssl/private/client-selfsigned.key
[rock@rockpi-4b:~/DUF_SERVER$ sudo ls -l /etc/ssl/certs/client-selfsigned.crt
 -rw-r--r-- 1 root root 1513 May 16 00:04 /etc/ssl/certs/client-selfsigned.crt
 rock@rockpi-4b:~/DUF_SERVER$ 
```

Figure 10: The DUF server key generation step to support encrypted communications using TLS

3.2.2.1. Commissioning

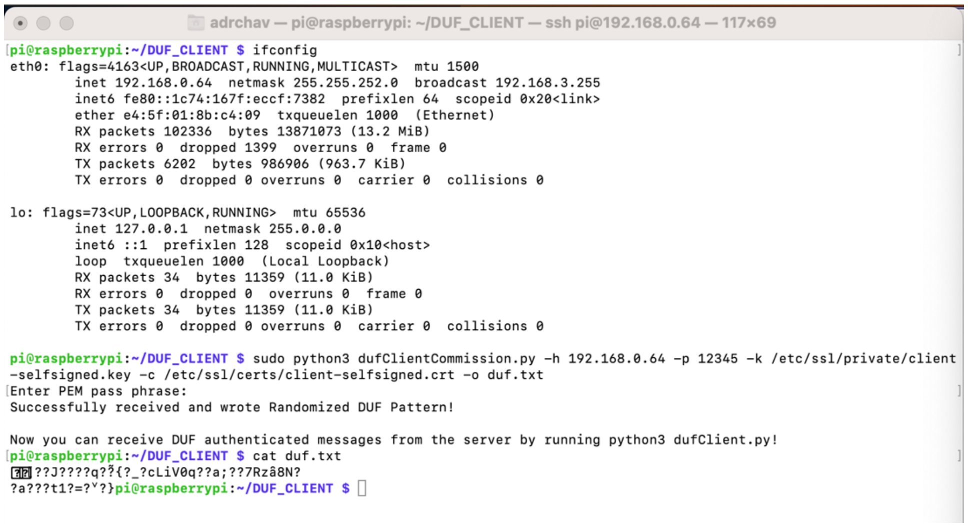
1. On the **client** start the commissioning python3 script `python3 dufClientCommission.py -h clientIP -p clientPort -k /path/to/private/key -c /path/to/certificate -o`

`/path/to/dufSecretKeyFile` (as shown below in Figure and Figure) where **clientIP** is the IP address of the client system (default 10.36.1.193) **clientPort** is the TCP port number of the client system (default 12345) **/path/to/private/key** is the path to the openssl private key file the client system (default /etc/ssl/private/client-selfsigned.key) **/path/to/certificate** is the path to the openssl certificate file the client system (default /etc/ssl/certs/client-selfsigned.crt) **/path/to/dufSecretKeyFile** is the key that would be placed in a secure, tamper-resistant storage space (default ./duf.txt) You will be prompted for the admin password initially and then the password for the private key. Ignore the deprecation warning.



```python
import socket
import ssl
import getopt
import sys

HOST = "10.36.1.193"
PORT = 12345
USAGE = f"Usage: python3 {sys.argv[0]} [-h host_ip] [-p host_port] [-k private_key] [-c cert_file] [-o outputfile]"


def parse(args):
    HOST="10.36.1.193"
    PORT = 12345
    KEY = "/etc/ssl/private/client-selfsigned.key"
    CERT = "/etc/ssl/certs/client-selfsigned.crt"
    OUTFILE = "duf.txt"

    options, arguments = getopt.getopt(args, 'h:p:k:c:o:', ["host=", "port=", "key=", "cert=", "out="])
    for o, a in options:
        if o in ("-h", "--host"):
            HOST = a
        if o in ("-p", "--port"):
            PORT = int(a)
        if o in ("-k", "--key"):
            KEY = a
        if o in ("-c", "--cert"):
            CERT = a
        if o in ("-o", "--out"):
            OUTFILE = a
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return (HOST, PORT, KEY, CERT, OUTFILE)


if __name__ == "__main__":
    args = sys.argv[1:]
    (HOST, PORT, KEY, CERT, OUTFILE) = parse(args)
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server = ssl.wrap_socket(server, server_side=True, keyfile=KEY, certfile=CERT)

    server.bind((HOST, PORT))
    server.listen(0)

    connection, client_address = server.accept()

    #while True:
    data = connection.recv(64)
    binary_file = open(OUTFILE, "wb")
    binary_file.write(data)
    binary_file.close()

    #    if not data:
    #        break
    print(f"Successfully received and wrote Randomized DUF Pattern!")
    print()
    print("Now you can receive DUF authenticated messages from the server by running python3 dufClient.py!")
```

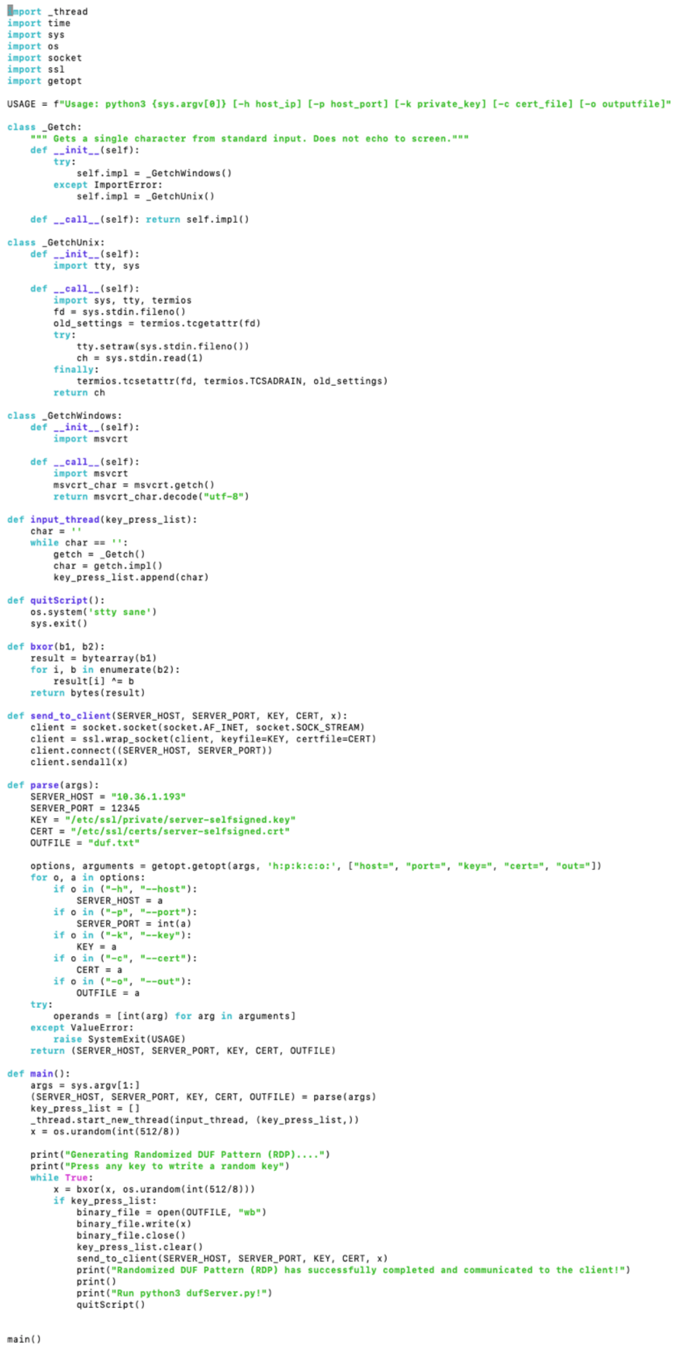
Figure 11. The DUF client commissioning python program



```
adrchav — pi@raspberrypi: ~/DUF_CLIENT — ssh pi@192.168.0.64 — 117×69
pi@raspberrypi:~/DUF_CLIENT $ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.64  netmask 255.255.252.0  broadcast 192.168.3.255
        inet6 fe80::1c74:167f:eccf:7382  prefixlen 64  scopeid 0x20<link>
        ether e4:5f:01:8b:c4:09  txqueuelen 1000  (Ethernet)
        RX packets 102336  bytes 13871073 (13.2 MiB)
        RX errors 0  dropped 1399  overruns 0  frame 0
        TX packets 6202  bytes 986906 (963.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 34  bytes 11359 (11.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 34  bytes 11359 (11.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

pi@raspberrypi:~/DUF_CLIENT $ sudo python3 dufClientCommission.py -h 192.168.0.64 -p 12345 -k /etc/ssl/private/client
-selfsigned.key -c /etc/ssl/certs/client-selfsigned.crt -o duf.txt
Enter PEM pass phrase:
Successfully received and wrote Randomized DUF Pattern!

Now you can receive DUF authenticated messages from the server by running python3 dufClient.py!
pi@raspberrypi:~/DUF_CLIENT $ cat duf.txt
??????J????q??{?_?cLiV0q??a;??7Rzâ8N?
?a???t1?=?ˇ?}pi@raspberrypi:~/DUF_CLIENT $ 
```

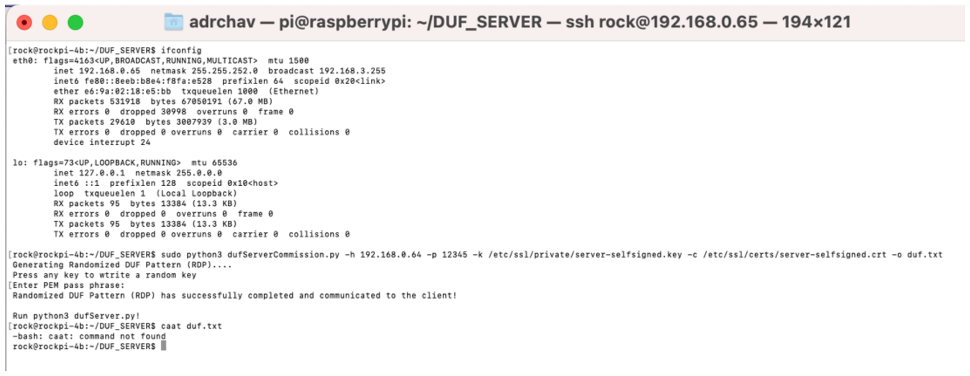
Figure 12: The DUF client results of the commissioning phase

2. On the **server** start the commissioning python3 script `python3 dufServerCommission.py -h clientIP -p clientPort -k /path/to/private/key -c /path/to/certificate -o /path/to/dufSecretKeyFile` (as shown below in Figure and Figure) where **clientIP** is the IP address of the client system (default 10.36.1.193) **clientPort** is the TCP port number of the client system (default 12345) **/path/to/private/key** is the path to the openssl private key file the server system (default /etc/ssl/private/server-selfsigned.key) **/path/to/certificate** is the path to the openssl certificate file the server system (default /etc/ssl/certs/server-selfsigned.crt) **/path/to/dufSecretKeyFile** is the key that would be placed in a secure, tamper-resistant storage space (default ./duf.txt) You will be prompted for the admin password initially and then the password for the private key. Ignore the deprecation warning.



```python
import _thread
import time
import sys
import os
import socket
import ssl
import getopt

USAGE = f"Usage: python3 {sys.argv[0]} [-h host_ip] [-p host_port] [-k private_key] [-c cert_file] [-o outputfile]"

class _Getch:
    """ Gets a single character from standard input. Does not echo to screen."""
    def __init__(self):
        try:
            self.impl = _GetchWindows()
        except ImportError:
            self.impl = _GetchUnix()

    def __call__(self): return self.impl()

class _GetchUnix:
    def __init__(self):
        import tty, sys

    def __call__(self):
        import sys, tty, termios
        fd = sys.stdin.fileno()
        old_settings = termios.tcgetattr(fd)
        try:
            tty.setraw(sys.stdin.fileno())
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
        return ch

class _GetchWindows:
    def __init__(self):
        import msvcrt

    def __call__(self):
        import msvcrt
        msvcrt_char = msvcrt.getch()
        return msvcrt_char.decode("utf-8")

def input_thread(key_press_list):
    char = ''
    while char == '':
        getch = _Getch()
        char = getch.impl()
        key_press_list.append(char)

def quitScript():
    os.system('stty sane')
    sys.exit()

def bxor(b1, b2):
    result = bytearray(b1)
    for i, b in enumerate(b2):
        result[i] ^= b
    return bytes(result)

def send_to_client(SERVER_HOST, SERVER_PORT, KEY, CERT, x):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client = ssl.wrap_socket(client, keyfile=KEY, certfile=CERT)
    client.connect((SERVER_HOST, SERVER_PORT))
    client.sendall(x)

def parse(args):
    SERVER_HOST = "10.36.1.193"
    SERVER_PORT = 12345
    KEY = "/etc/ssl/private/server-selfsigned.key"
    CERT = "/etc/ssl/certs/server-selfsigned.crt"
    OUTFILE = "duf.txt"

    options, arguments = getopt.getopt(args, 'h:p:k:c:o:', ["host=", "port=", "key=", "cert=", "out="])
    for o, a in options:
        if o in ("-h", "--host"):
            SERVER_HOST = a
        if o in ("-p", "--port"):
            SERVER_PORT = int(a)
        if o in ("-k", "--key"):
            KEY = a
        if o in ("-c", "--cert"):
            CERT = a
        if o in ("-o", "--out"):
            OUTFILE = a
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return (SERVER_HOST, SERVER_PORT, KEY, CERT, OUTFILE)

def main():
    args = sys.argv[1:]
    (SERVER_HOST, SERVER_PORT, KEY, CERT, OUTFILE) = parse(args)
    key_press_list = []
    _thread.start_new_thread(input_thread, (key_press_list,))
    x = os.urandom(int(512/8))

    print("Generating Randomized DUF Pattern (RDP)....")
    print("Press any key to wtrite a random key")
    while True:
        x = bxor(x, os.urandom(int(512/8)))
        if key_press_list:
            binary_file = open(OUTFILE, "wb")
            binary_file.write(x)
            binary_file.close()
            key_press_list.clear()
            send_to_client(SERVER_HOST, SERVER_PORT, KEY, CERT, x)
            print("Randomized DUF Pattern (RDP) has successfully completed and communicated to the client!")
            print()
            print("Run python3 dufServer.py!")
            quitScript()


main()
```

Figure 13: The DUF server commissioning python program


adrchav — pi@raspberrypi: ~/DUF_SERVER — ssh rock@192.168.0.65 — 194×121

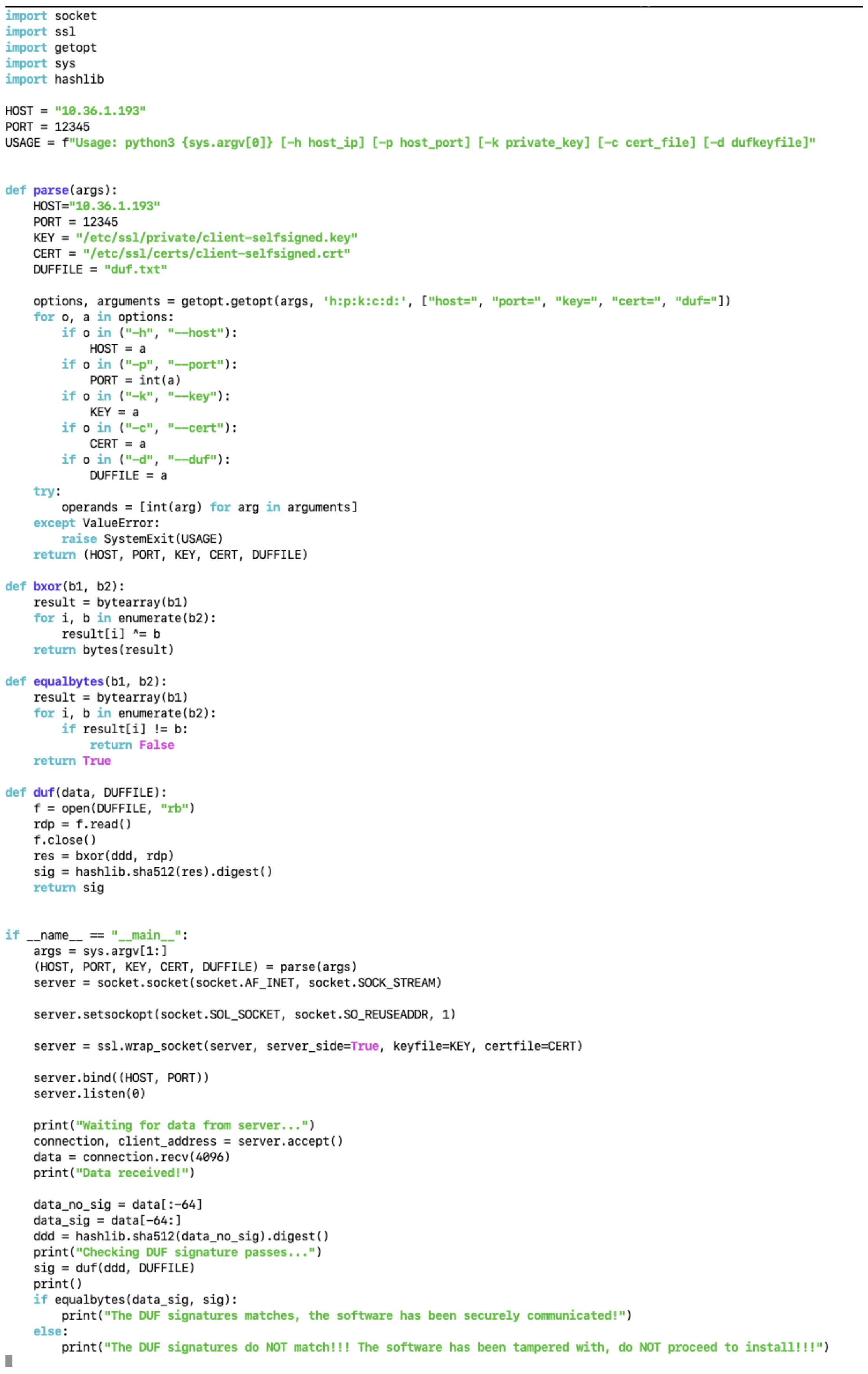
```
[rock@rockpi-4b:~/DUF_SERVER$ ifconfig
 eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
         inet 192.168.0.65  netmask 255.255.252.0  broadcast 192.168.3.255
         inet6 fe80::8eeb:b8e4:f8fa:e528  prefixlen 64  scopeid 0x20<link>
         ether e6:9a:02:18:e5:bb  txqueuelen 1000  (Ethernet)
         RX packets 531918  bytes 67050191 (67.0 MB)
         RX errors 0  dropped 30998  overruns 0  frame 0
         TX packets 29610  bytes 3007939 (3.0 MB)
         TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
         device interrupt 24

 lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
         inet 127.0.0.1  netmask 255.0.0.0
         inet6 ::1  prefixlen 128  scopeid 0x10<host>
         loop  txqueuelen 1  (Local Loopback)
         RX packets 95  bytes 13384 (13.3 KB)
         RX errors 0  dropped 0  overruns 0  frame 0
         TX packets 95  bytes 13384 (13.3 KB)
         TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[rock@rockpi-4b:~/DUF_SERVER$ sudo python3 dufServerCommission.py -h 192.168.0.64 -p 12345 -k /etc/ssl/private/server-selfsigned.key -c /etc/ssl/certs/server-selfsigned.crt -o duf.txt
 Generating Randomized DUF Pattern (RDP)....
 Press any key to wtrite a random key
[Enter PEM pass phrase:
 Randomized DUF Pattern (RDP) has successfully completed and communicated to the client!

 Run python3 dufServer.py!
[rock@rockpi-4b:~/DUF_SERVER$ caat duf.txt
 -bash: caat: command not found
 rock@rockpi-4b:~/DUF_SERVER$ █
```

Figure 14: The DUF server results of the commissioning phase

3. Both client and server should report a success message and there should also be a file named duf.txt in the current working directory. This file contains a binary 512-bit cryptographic key (as shown in Figure and Figure).

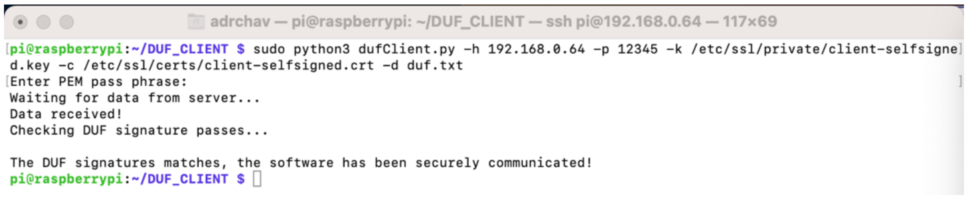
3.2.2.2. Operating

1. On the **client** start the operating python3 script `python3 dufClient.py -h clientIP -p clientPort -k /path/to/private/key -c /path/to/certificate -d /path/to/dufSecretKeyFile` (as shown below in Figure 4 and Figure 5) where **clientIP** is the IP address of the client system (default 10.36.1.193) **clientPort** is the TCP port number of the client system (default 12345) **/path/to/private/key** is the path to the openssl private key file the client system (default /etc/ssl/private/client-selfsigned.key) **/path/to/certificate** is the path to the openssl certificate file the client system (default /etc/ssl/certs/client-selfsigned.crt) **/path/to/dufSecretKeyFile** is the key that would be placed in a secure, tamper-resistant storage space (default ./duf.txt) You will be prompted for the admin password initially and then the password for the private key. Ignore the deprecation warning.



```python
import socket
import ssl
import getopt
import sys
import hashlib

HOST = "10.36.1.193"
PORT = 12345
USAGE = f"Usage: python3 {sys.argv[0]} [-h host_ip] [-p host_port] [-k private_key] [-c cert_file] [-d dufkeyfile]"


def parse(args):
    HOST="10.36.1.193"
    PORT = 12345
    KEY = "/etc/ssl/private/client-selfsigned.key"
    CERT = "/etc/ssl/certs/client-selfsigned.crt"
    DUFFILE = "duf.txt"

    options, arguments = getopt.getopt(args, 'h:p:k:c:d:', ["host=", "port=", "key=", "cert=", "duf="])
    for o, a in options:
        if o in ("-h", "--host"):
            HOST = a
        if o in ("-p", "--port"):
            PORT = int(a)
        if o in ("-k", "--key"):
            KEY = a
        if o in ("-c", "--cert"):
            CERT = a
        if o in ("-d", "--duf"):
            DUFFILE = a
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return (HOST, PORT, KEY, CERT, DUFFILE)

def bxor(b1, b2):
    result = bytearray(b1)
    for i, b in enumerate(b2):
        result[i] ^= b
    return bytes(result)

def equalbytes(b1, b2):
    result = bytearray(b1)
    for i, b in enumerate(b2):
        if result[i] != b:
            return False
    return True

def duf(data, DUFFILE):
    f = open(DUFFILE, "rb")
    rdp = f.read()
    f.close()
    res = bxor(ddd, rdp)
    sig = hashlib.sha512(res).digest()
    return sig


if __name__ == "__main__":
    args = sys.argv[1:]
    (HOST, PORT, KEY, CERT, DUFFILE) = parse(args)
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server = ssl.wrap_socket(server, server_side=True, keyfile=KEY, certfile=CERT)

    server.bind((HOST, PORT))
    server.listen(0)

    print("Waiting for data from server...")
    connection, client_address = server.accept()
    data = connection.recv(4096)
    print("Data received!")

    data_no_sig = data[:-64]
    data_sig = data[-64:]
    ddd = hashlib.sha512(data_no_sig).digest()
    print("Checking DUF signature passes...")
    sig = duf(ddd, DUFFILE)
    print()
    if equalbytes(data_sig, sig):
        print("The DUF signatures matches, the software has been securely communicated!")
    else:
        print("The DUF signatures do NOT match!!! The software has been tampered with, do NOT proceed to install!!!")
```

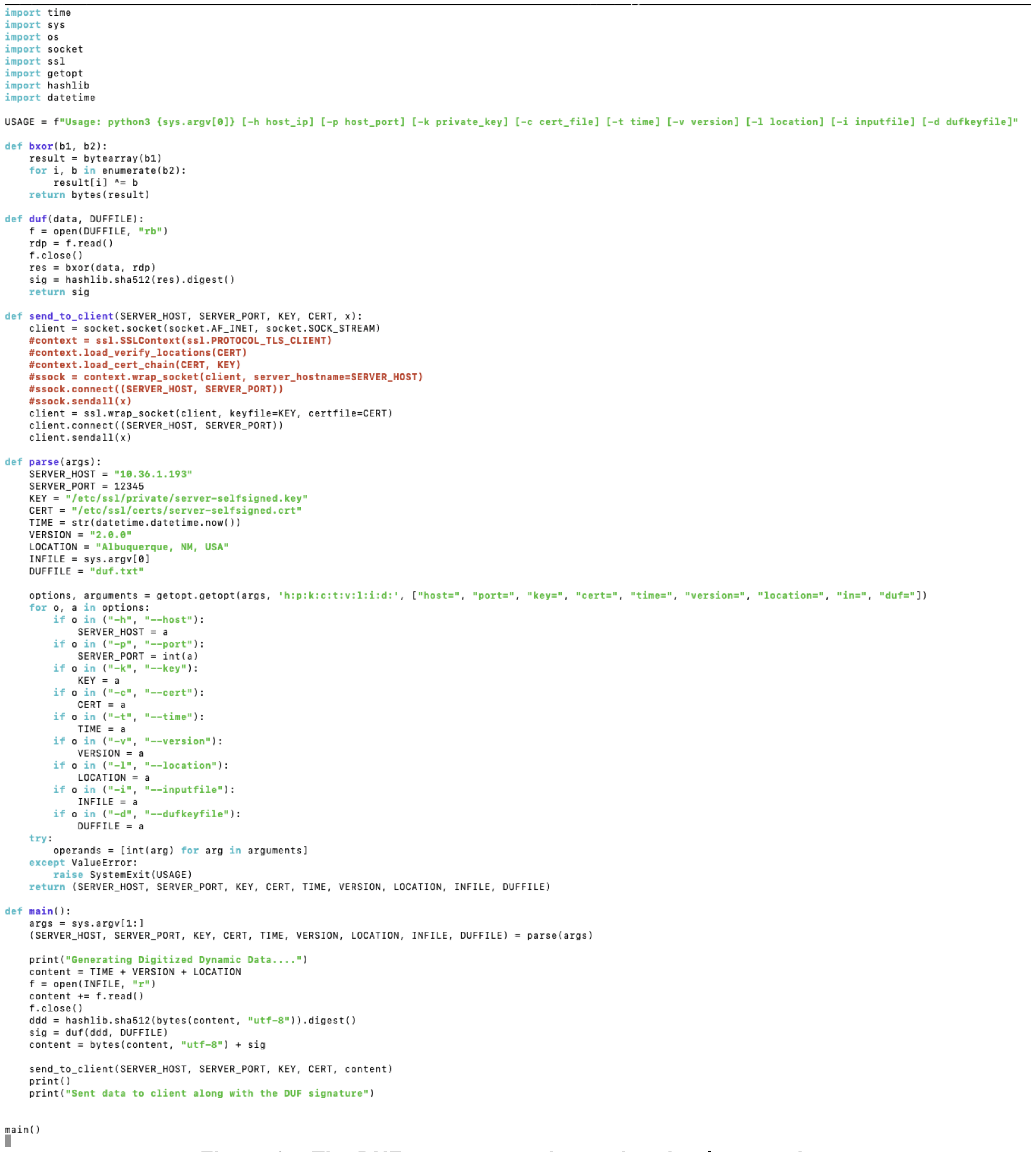
Figure 45: The DUF client operation python implementation



```
pi@raspberrypi:~/DUF_CLIENT $ sudo python3 dufClient.py -h 192.168.0.64 -p 12345 -k /etc/ssl/private/client-selfsigne
d.key -c /etc/ssl/certs/client-selfsigned.crt -d duf.txt
Enter PEM pass phrase:
Waiting for data from server...
Data received!
Checking DUF signature passes...

The DUF signatures matches, the software has been securely communicated!
pi@raspberrypi:~/DUF_CLIENT $ 
```

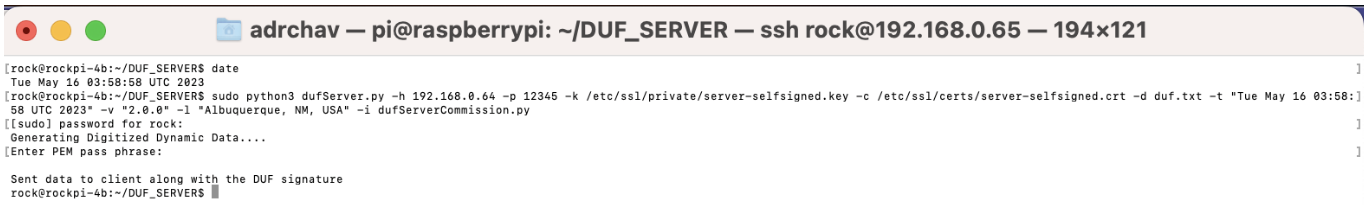
Figure 56: The results of the operation phase of a file validated by the DUF client

2. On the **server** start the operating python3 script `python3 dufServer.py -h clientIP -p clientPort -k /path/to/private/key -c /path/to/certificate -t timestamp -v version -l location -i /path/to/inputfile -d /path/to/dufSecretKeyFile` (as shown below in Figure 6 and Figure 7) where **clientIP** is the IP address of the client system (default 10.36.1.193) **clientPort** is the TCP port number of the client system (default 12345) **/path/to/private/key** is the path to the openssl private key file the server system (default /etc/ssl/private/server-selfsigned.key)**/path/to/certificate** is the path to the openssl certificate file the server system (default /etc/ssl/certs/server-selfsigned.crt) **timestamp** is a string of the timestamp used for the meta-data transferred to the client (default is current date and time) **version** is a string of the version number of the data that will be sent to the client which is part of the meta-data transferred to the client (default "2.0.0") **location** is a string of the physical location of the data that will be sent to the client which is part of the meta-data transferred to the client (default "Albuquerque, NM, USA") **/path/to/inputdata** is the file of data that will be transferred to the client (default is this program "dufServer.py") **/path/to/dufSecretKeyFile** is the key that would be placed in a secure, tamper-resistant storage space (default ./duf.txt) You will be prompted for the admin password initially and then the password for the private key. Ignore the deprecation warning.



```python
import time
import sys
import os
import socket
import ssl
import getopt
import hashlib
import datetime

USAGE = f"Usage: python3 {sys.argv[0]} [-h host_ip] [-p host_port] [-k private_key] [-c cert_file] [-t time] [-v version] [-l location] [-i inputfile] [-d dufkeyfile]"

def bxor(b1, b2):
    result = bytearray(b1)
    for i, b in enumerate(b2):
        result[i] ^= b
    return bytes(result)

def duf(data, DUFFILE):
    f = open(DUFFILE, "rb")
    rdp = f.read()
    f.close()
    res = bxor(data, rdp)
    sig = hashlib.sha512(res).digest()
    return sig

def send_to_client(SERVER_HOST, SERVER_PORT, KEY, CERT, x):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    #context.load_verify_locations(CERT)
    #context.load_cert_chain(CERT, KEY)
    #ssock = context.wrap_socket(client, server_hostname=SERVER_HOST)
    #ssock.connect((SERVER_HOST, SERVER_PORT))
    #ssock.sendall(x)
    client = ssl.wrap_socket(client, keyfile=KEY, certfile=CERT)
    client.connect((SERVER_HOST, SERVER_PORT))
    client.sendall(x)

def parse(args):
    SERVER_HOST = "10.36.1.193"
    SERVER_PORT = 12345
    KEY = "/etc/ssl/private/server-selfsigned.key"
    CERT = "/etc/ssl/certs/server-selfsigned.crt"
    TIME = str(datetime.datetime.now())
    VERSION = "2.0.0"
    LOCATION = "Albuquerque, NM, USA"
    INFILE = sys.argv[0]
    DUFFILE = "duf.txt"

    options, arguments = getopt.getopt(args, 'h:p:k:c:t:v:l:i:d:', ["host=", "port=", "key=", "cert=", "time=", "version=", "location=", "in=", "duf="])
    for o, a in options:
        if o in ("-h", "--host"):
            SERVER_HOST = a
        if o in ("-p", "--port"):
            SERVER_PORT = int(a)
        if o in ("-k", "--key"):
            KEY = a
        if o in ("-c", "--cert"):
            CERT = a
        if o in ("-t", "--time"):
            TIME = a
        if o in ("-v", "--version"):
            VERSION = a
        if o in ("-l", "--location"):
            LOCATION = a
        if o in ("-i", "--inputfile"):
            INFILE = a
        if o in ("-d", "--dufkeyfile"):
            DUFFILE = a
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return (SERVER_HOST, SERVER_PORT, KEY, CERT, TIME, VERSION, LOCATION, INFILE, DUFFILE)

def main():
    args = sys.argv[1:]
    (SERVER_HOST, SERVER_PORT, KEY, CERT, TIME, VERSION, LOCATION, INFILE, DUFFILE) = parse(args)

    print("Generating Digitized Dynamic Data....")
    content = TIME + VERSION + LOCATION
    f = open(INFILE, "r")
    content += f.read()
    f.close()
    ddd = hashlib.sha512(bytes(content, "utf-8")).digest()
    sig = duf(ddd, DUFFILE)
    content = bytes(content, "utf-8") + sig

    send_to_client(SERVER_HOST, SERVER_PORT, KEY, CERT, content)
    print()
    print("Sent data to client along with the DUF signature")


main()
```

Figure 67: The DUF server operation python implementation



adrchav — pi@raspberrypi: ~/DUF_SERVER — ssh rock@192.168.0.65 — 194×121

```
rock@rockpi-4b:~/DUF_SERVER$ date
Tue May 16 03:58:58 UTC 2023
rock@rockpi-4b:~/DUF_SERVER$ sudo python3 dufServer.py -h 192.168.0.64 -p 12345 -k /etc/ssl/private/server-selfsigned.key -c /etc/ssl/certs/server-selfsigned.crt -d duf.txt -t "Tue May 16 03:58:
58 UTC 2023" -v "2.0.0" -l "Albuquerque, NM, USA" -i dufServerCommission.py
[sudo] password for rock:
Generating Digitized Dynamic Data....
Enter PEM pass phrase:

Sent data to client along with the DUF signature
rock@rockpi-4b:~/DUF_SERVER$ 
```

Figure 78: The results of the operation phase of a file validated by the DUF server

3. Both client and server should report a success message and there should be a message on the client that the DUF signature matched as shown in Figure 5 and Figure 7. This step completes the demonstration that the client can successfully validate the DUF signature of data transferred from a server system.

4. CONCLUSION AND FUTURE DIRECTIONS

The DUF technology being matured and evaluated was awarded US Patent # 10,541,996 B1. Sandia National Laboratory and Active Assurance Inc. entered into an exclusive license agreement effective June 22, 2021, under which Active Assurance can commercialize the technology. The technology was selected by Sandia National Laboratories to compete in the 2021 R&D Awards competition, which was successfully won by the team.

The DUF technology is a zero-trust tamper-resistant high-assurance session-less dynamic and active device authentication protocol that simultaneously authenticates identity and provides integrity verification in a single step, substantially reducing the risk of cyberattack, and eliminating the need for costly and complex conventional communication security systems requirements (i.e., cryptography, PKI, and key management).

Preliminary proof of concept software and hardware have been developed. The technology requires additional software and hardware development, an independent security assessment, and validation testing. The prototype software does not leverage a truly tamper-resistant hardware device for the RDP. The technology may also require additional certification for use in regulated industries. Maturation outcomes include a laboratory validated technology and operational prototypes tested in a real-world environment.

Several prospective strategic partners and commercial customers have expressed interest in the technology, and the expected development and maturation of the technology will greatly advance the commercialization of the technology and the opportunity to raise capital.