



BURSA TEKNİK ÜNİVERSİTESİ

BİLGİSAYAR AĞLARI ARA RAPORU

**Proje Konusu: Python ile Geliştirilen Temel Güvenlikli Dosya Paylaşım Sistemi:
Şifreleme, Paket Parçalama ve Ağ Trafiği Analizi Uygulaması**

AD-SOYAD: Adile AKKILIÇ

NUMARASI: 21360859052

DANIŞMAN AD SOYAD: Doç. Dr. İzzet Fatih ŞENTÜRK

1. INTRODUCTION

1.1. Problem Tanımı ve Motivasyon

Bugünlerde dosya transferi işleri büyük veri ve artan siber tehditler yüzünden eski araçların ötesinde güvenlik ve esneklik istiyor. Birçok öğrenci, IP başlığındaki bayraklar, TTL, parçalama ve hata denetimi gibi düşük seviye detayları doğrudan tecrübe etme fırsatını bulmadan mezun oluyor. Bu proje tam da bu boşluğu doldurmak için şifreli AES-256 ve elle hazırlanmış IP paketleri kullanan bir dosya aktarım sistemi geliştirerek hem güvenliği artırmayı hemde ağ katmanının iç yüzünü pratikte göstermeyi amaçlıyor.

1.2. Projenin Amacı & Kapsamı

Proje, iki Linux makine arasında güvenli ve düşük seviye kontrolle herhangi bir boyutta dosya transferi gerçekleştirebilecek bir araç takımı geliştirmeyi hedeflemektedir. Bu kapsamda:

- Dosyaların belirli bir sabit boyuta bölünmesi ve Python tabanlı istemci/sunucu kodu aracılığıyla her bir parçanın şifrelenmesi ve alıcı tarafta yeniden birleştirme.
- Scapy ile IPv4 başlığı modülasyonu, ve başlık ofsetlerinizde DF/MF bayrakları, kontrol yüz, ve TTL alanlarının manuel değiştirilmesi.
- SHA-256 özeti olan her parça için bütünlük denetimi ve paketin alıcıya yanlış gidiyorsa yeniden gönderilmesi.
- tc aracı ile RTT, gecikme, jitter ve paket kaybı ekleyerek, bant genişliği, paket kaybı gibi performans metriklerin ölçülmesi.
- Wireshark ve mitmproxy ile bir saldırı durumunda, paket içeriğinin okunamadığının doğrulanması.

1.3. Bugüne Kadarki İlerleme

- **Ortam kurulumu:** VirtualBox üzerinde Ubuntu 22.04, Python 3.12, Scapy 2.5.0, pycryptodome ve Wireshark.
- **Parça tabanlı aktarım prototipi:** client_chunk.py / server_chunk.py ile 100 KB-5 MB dosyalar TCP 5000 üzerinden başarıyla iletildi (ör. 100 KB \approx 0,15 sn).
- **TTL varyantı:** client_chunk_ttl.py dosyası her pakete ayarlanabilir TTL ekliyor; 5 MB dosya (641 parça) \sim 22 sn'de ulaşıyor.
- **Şifreleme katmanı:** AES-256-CBC + SHA-256 entegrasyonu; düz ve TTL modlarında aynı hash çıktıları elde edildi (örn. c036cbb7...c3887e29).
- **Performans testleri:** Disk çıkışı \sim 41 MB/s, 100 KB dosyalarda $<$ 150 ms aktarım; farklı dosya boyutlarında parça sayıları ve süreler kaydedildi.
- **Paket yakalama:** Wireshark ekran görüntüsünde (Şekil 1) özel RSL protokolü “Malformed Packet” olarak işaretleniyor—beklenen bir sonuç, çünkü standart dışı yük yapısı kullanılıyor.

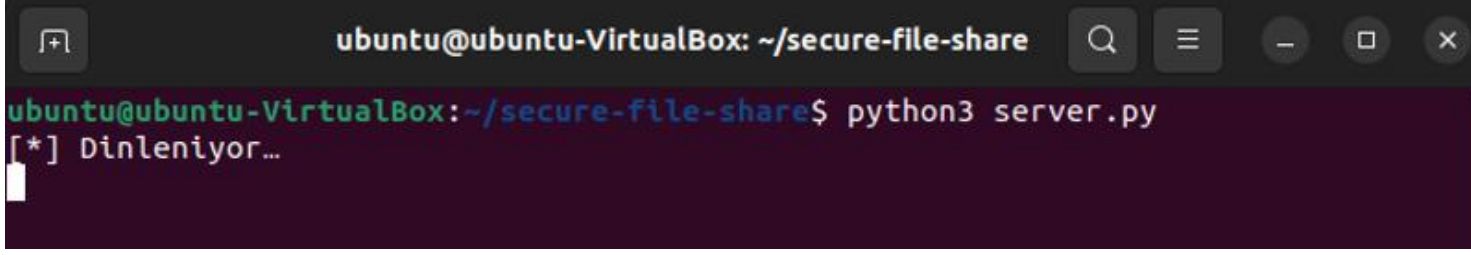
2. TECHNICAL DETAILS

2.1. File Transfer Module (TCP)

Bu katman, tek seferde şifreli veri aktaran client.py (gönderici) ve server.py (alıcı) betiklerinden oluşur. Aşağıda, iki uçtaki akış ayrıntılı olarak sunulmuş, ilgili terminal ekranları (Şekil 2-1a–c) numaralı görsellerle ilişkilendirilmiştir.

Bağlantı Kurulumu: Sunucu, socket() nesnesini TCP 5000 portuna bağlayarak dinleme konumuna geçer ve

beklediği bağlantıyı kabul eder. Bu evre sonunda terminalde “[*] Dinleniyor...” ve istemcinin bağlanmasıyla “[+] Bağlandı (adres, port)” ifadeleri görülür (Şekil 2-1a).



```
ubuntu@ubuntu-VirtualBox: ~/secure-file-share
ubuntu@ubuntu-VirtualBox:~/secure-file-share$ python3 server.py
[*] Dinleniyor...
```

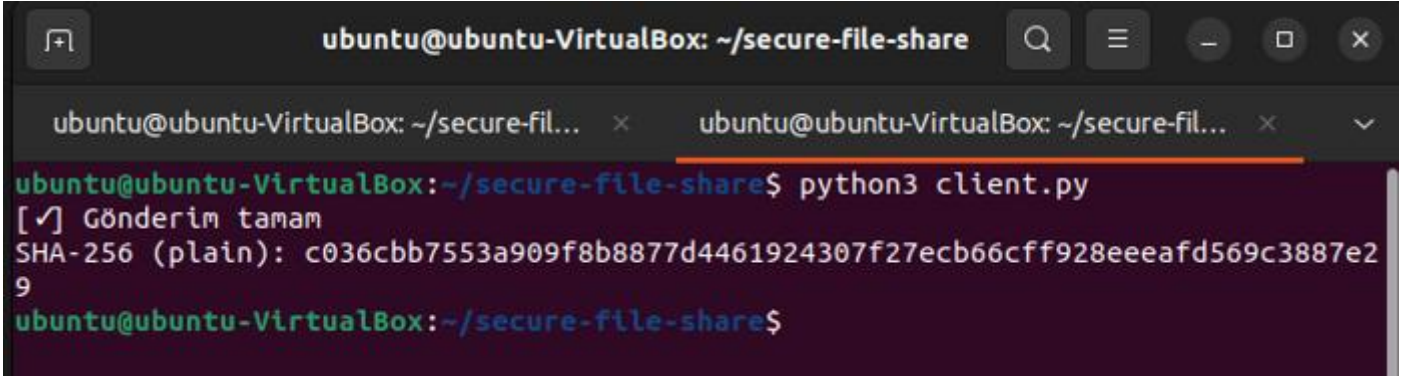
Şekil 2-1a : Sunucu dinleniyor...

Uzunluk Ön-Başlığı: İstemci, şifreli yükün tam bayt uzunluğunu `struct.pack("I", len(ciphertext))` ile 4 bayt, büyük-endian biçiminde gönderir. Sunucu gelen ilk 4 baytı okuyarak bekleyeceği veri miktarını kesin olarak belirler; böylece veri sonuna kadar alınmadan akış sonu kabul edilmez.

Veri Akışı: Uzunluk bilgisi alındıktan sonra istemci, şifreli içeriğini tek çağrıda `sendall()` ile gönderir. Sunucu ise bir `recv()` döngüsü içerisinde belirlenen uzunluğa ulaşana dek veriyi almaya devam eder. Veri alımı sırasında ara alınan bayt miktarları terminale yazdırılmadan, doğrudan tüm veri tamamlanana kadar süreç sürdürülür.

Şifre Çözme ve Dolgu Temizleme: Sunucu veriyi tam aldığı anda, AES-256-CBC kipinde aynı anahtar-IV ikilisiyle çözüm işlemini gerçekleştirir. Son baytta tutulan dolgu uzunluğu okunarak PKCS-7 dolgu temizlenir; böylece özgün düz metin elde edilir.

Bütünlük Doğrulaması: İstemci gönderimden önce, sunucu ise çözümden sonra düz verinin SHA-256 özetini üretir ve çıktıya yazar. Her iki uçtaki özet değerlerinin bire bir eşleşmesi, aktarım sırasında bit hatası veya paket kaybı olmadığına kanıttır (Şekil 2-1b).



```
ubuntu@ubuntu-VirtualBox: ~/secure-file-share
ubuntu@ubuntu-VirtualBox:~/secure-file-share$ python3 client.py
[✓] Gönderim tamam
SHA-256 (plain): c036cbb7553a909f8b8877d4461924307f27ecb66cff928eeeaafd569c3887e2
9
ubuntu@ubuntu-VirtualBox:~/secure-file-share$
```

(Şekil 2-1b) : İstemci gönderimi tamamlanması ve SHA-256 hash göstermesi

Dosyanın Kalıcı Saklanması: Doğrulama başarıyla tamamlandığında, sunucu içeriği `received.txt` adına kaydederek oturumu sonlandırır; istemci tarafında ise “[✓] Gönderim tamam” ile birlikte işlem süresi ve özet değeri görüntülenir.

```
ubuntu@ubuntu-VirtualBox: ~/secure-file-share
ubuntu@ubuntu-VirtualBox: ~/secure-file-share$ python3 server.py
[*] Dinleniyor...
[+] Bağlandı: ('127.0.0.1', 46464)
[✓] Çözüm tamam
SHA-256 (plain): c036cbb7553a909f8b8877d4461924307f27ecb66cff928eeeafd569c3887e29
ubuntu@ubuntu-VirtualBox: ~/secure-file-share$
```

Şekil 2-1c : Şekil 2-1a (bağlantı kabulü) + Şekil 2-1b (sunucu tarafı)

(Şekil 2-1a) Sunucuda bekleme ve bağlantı kabulü.

(Şekil 2-1b) İstemci ve sunucuda eşleşen SHA-256 çıktıları ile “Gönderim tamam” bildirimi.

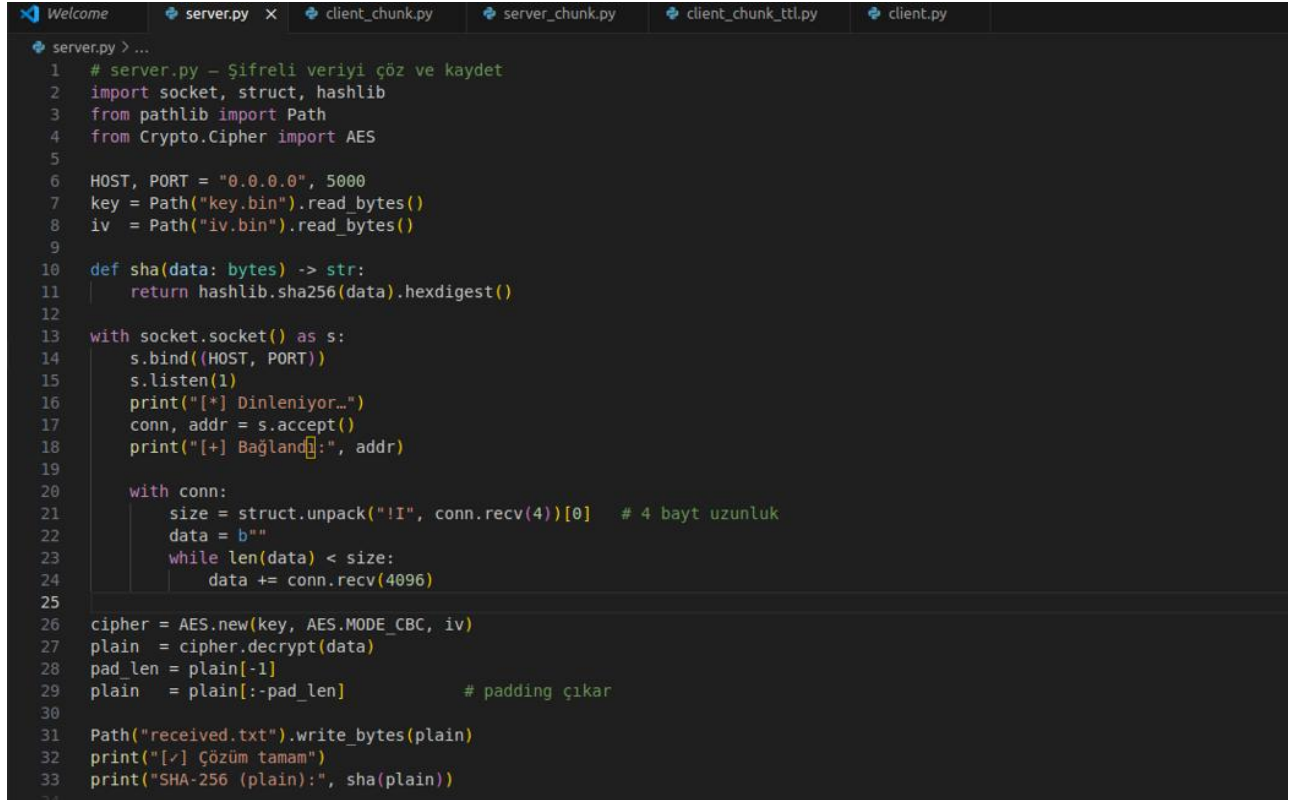
Bu yapı, tek parça aktarım için düşük başlık yükü ve sade hata denetimi (uzunluk + özet) sağlayarak projenin daha karmaşık parça-tabanlı modüllerine temel oluşturur.

```
Welcome x server.py client_chunk.py server_chunk.py client_chunk_ttl.py client.py x
client.py > ...
3 from pathlib import Path
4 from Crypto.Cipher import AES
5
6 HOST, PORT = "127.0.0.1", 5000
7 FILE = Path("sample.txt")
8 key = Path("key.bin").read_bytes()
9 iv = Path("iv.bin").read_bytes()
10
11 def sha(path: Path) -> str:
12     return hashlib.sha256(path.read_bytes()).hexdigest()
13
14 # 1) Dosyayı oku ve 16 bayta pad et
15 plain = FILE.read_bytes()
16 pad_len = 16 - len(plain) % 16
17 plain += bytes([pad_len] * pad_len) # PKCS-7 çeşidi
18
19 # 2) Şifrele
20 cipher = AES.new(key, AES.MODE_CBC, iv)
21 ciphertext = cipher.encrypt(plain)
22
23 with socket.socket() as s:
24     s.connect((HOST, PORT))
25     # 3) Önce uzunluğu (4 bayt) yolla, sonra veriyi
26     s.sendall(struct.pack("I", len(ciphertext)))
27     s.sendall(ciphertext)
28
29 print("[✓] Gönderim tamam")
30 print("SHA-256 (plain):", sha(FILE))
```

Şekil 2-1d : client.py kodu

Şekil 2-1d’de görülen client.py kodu, bir dosyayı şifreleyerek TCP üzerinden alıcıya güvenli şekilde iletmek amacıyla tasarlanmıştır. Betik çalıştırıldığında, ilk olarak Path("key.bin").read_bytes() ve Path("iv.bin").read_bytes() fonksiyonları ile AES-256-CBC algoritması için gerekli anahtar ve IV (initialization vector) bilgileri dosyalardan okunur. Bu dosyalar, şifreleme işleminin güvenli olması için gizli tutulmalıdır. Ardından, şifrelenmek istenen dosya (sample.txt), Path(FILE).read_bytes() fonksiyonu ile ikili (binary) modda belleğe alınır. Eğer dosya boyutu AES’in 16 baytlık blok boyutunun katı değilse, kalan kısmı tamamlamak amacıyla sonuna PKCS#7 standardına uygun padding eklenir. Dolgu işlemi sırasında, eksik kalan bayt sayısı kadar padding karakteri (chr(padding)) veri sonuna eklenir. Veri hazırlandıktan sonra, AES.new(key, AES.MODE_CBC, iv) fonksiyonu ile bir AES şifreleyici nesnesi oluşturulur ve cipher.encrypt(plain) fonksiyonu kullanılarak verinin şifrenmesi sağlanır. Ortaya çıkan ciphertext değişkeni, şifreli veri bütünü temsil eder. Bağlantı aşamasında, socket() fonksiyonu kullanılarak TCP tipi bir soket nesnesi oluşturulur. connect() fonksiyonu yardımıyla bu soket sunucuya (127.0.0.1 adresi ve 5000 portu) bağlanır. Veri aktarımı

başlamadan önce, şifreli verinin uzunluğu `struct.pack("!I", len(ciphertext))` fonksiyonu kullanılarak dört baytlık bir ön başlık halinde hazırlanır. Burada `struct.pack` fonksiyonu, veriyi "büyük endian" sıralamada (en önemli bayt başta) kodlar ve böylece sunucuya verinin tam boyutu net şekilde bildirilir. Ardından, şifreli veri `sendall()` fonksiyonu ile kesintisiz bir şekilde soket üzerinden gönderilir. Gönderim tamamlandıktan sonra, dosyanın özgün halinin (şifrelenmeden önceki düz veri) SHA-256 özeti `sha256()` fonksiyonu ile hesaplanır ve `hexdigest()` fonksiyonu yardımıyla okunabilir bir biçimde terminale yazdırılır. Bu özet değeri, alıcı tarafın aldığı verinin doğruluğunu teyit etmek için kullanılır. İşlem başarıyla tamamlandığında, "[✓] Gönderim tamam" mesajı terminale basılır ve bağlantı kapatılarak sonlandırılır.



```
1 # server.py - Şifreli veriyi çöz ve kaydet
2 import socket, struct, hashlib
3 from pathlib import Path
4 from Crypto.Cipher import AES
5
6 HOST, PORT = "0.0.0.0", 5000
7 key = Path("key.bin").read_bytes()
8 iv = Path("iv.bin").read_bytes()
9
10 def sha(data: bytes) -> str:
11     return hashlib.sha256(data).hexdigest()
12
13 with socket.socket() as s:
14     s.bind((HOST, PORT))
15     s.listen(1)
16     print("[*] Dinleniyor...")
17     conn, addr = s.accept()
18     print("[+] Bağlandı:", addr)
19
20     with conn:
21         size = struct.unpack("!I", conn.recv(4))[0] # 4 bayt uzunluk
22         data = b""
23         while len(data) < size:
24             data += conn.recv(4096)
25
26 cipher = AES.new(key, AES.MODE_CBC, iv)
27 plain = cipher.decrypt(data)
28 pad_len = plain[-1]
29 plain = plain[:-pad_len] # padding çıkar
30
31 Path("received.txt").write_bytes(plain)
32 print("[✓] Çözüm tamam")
33 print("SHA-256 (plain):", sha(plain))
```

Şekil 2-1e : server.py kodu

Şekil 2-1e'deki server.py kodu, istemciden gelen şifreli veriyi doğru şekilde alıp çözmek ve kalıcı olarak kaydetmek için hazırlanmıştır. Çalıştırıldığında, `socket()` fonksiyonu ile TCP tipinde bir soket nesnesi oluşturur ve `bind()` fonksiyonu ile 0.0.0.0 adresine ve 5000 portuna bağlanarak dinleme moduna geçer. `listen()` fonksiyonu ile gelen bağlantıları beklemeye başlar ve `accept()` fonksiyonu yardımıyla istemciden gelen bağlantı talebini kabul eder. Bağlantı kurulduktan sonra, alıcı taraf `recv(4)` fonksiyonunu kullanarak ilk gelen 4 baytı okur ve `struct.unpack("!I", ...)` fonksiyonu yardımıyla şifreli verinin uzunluğunu çözer. Bu sayede, alınması gereken veri miktarı önceden kesin olarak bilinir. Daha sonra bir döngü içinde `recv(4096)` çağrıları ile veriler parça parça alınır ve tam veri boyutuna ulaşılan kadar biriktirilir. Veri alındıktan sonra, `AES.new(key, AES.MODE_CBC, iv)` fonksiyonu ile AES çözümleyici nesnesi oluşturulur ve `cipher.decrypt(data)` fonksiyonu ile şifreli veri çözülür. Çözüm sonrasında, PKCS#7 dolgusunun kaldırılması için alınan düz verinin son baytı kontrol edilir; burada bulunan sayı kadar son bayt çıkarılarak orijinal düz veri (plain) elde edilir. Elde edilen veri, `Path("received.txt").write_bytes(plain)` fonksiyonu kullanılarak disk üzerinde "received.txt" adlı bir dosyaya kaydedilir. Böylece verinin fiziksel olarak saklanması sağlanmış olur. Son adımda, alınan düz verinin SHA-256

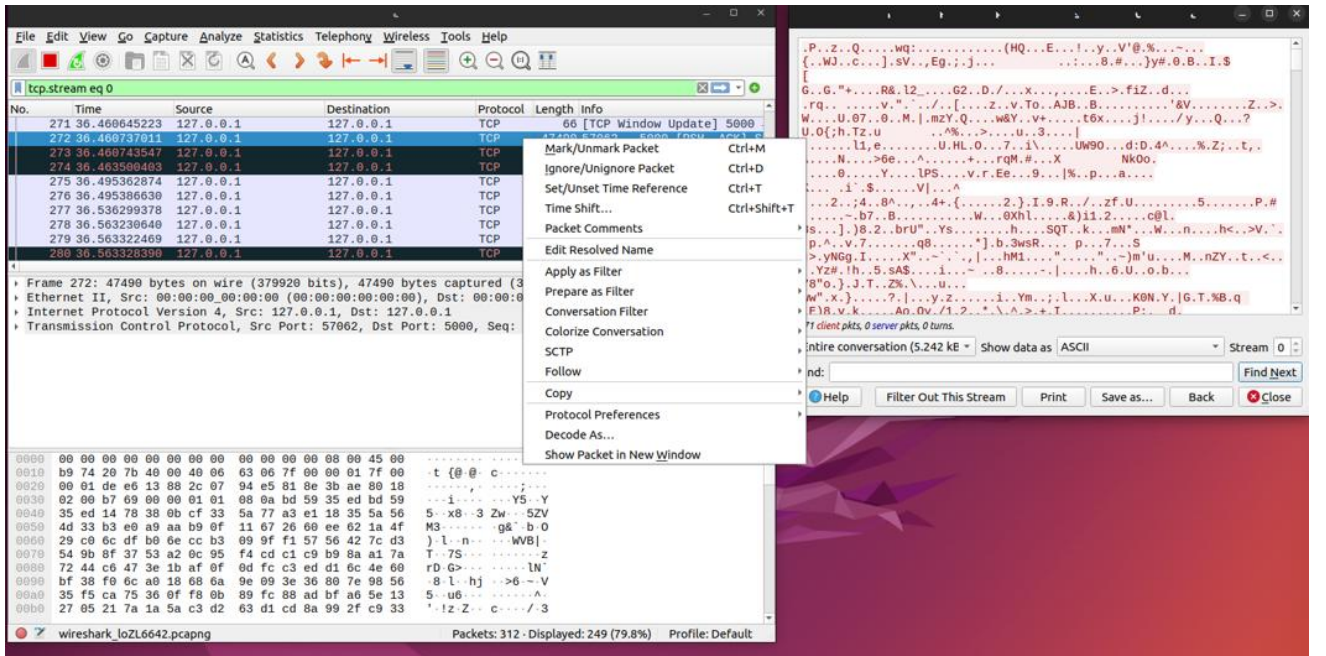
özet sha256() fonksiyonu ile hesaplanır ve hexdigest() fonksiyonu ile okunabilir formata çevrilerek terminal ekranına yazdırılır. İstemciden gönderilen özetle bu özet karşılaştırılarak verinin eksiksiz alınıp alınmadığı kontrol edilir. İşlemler tamamlandığında bağlantı kapatılır ve sunucu betiği sonlandırılır.

2.2. AES Encryption Layer

Bu katman, hem gizlilik hem de bütünlük sağlamak üzere iki aşamalı bir mekanizmaya dayanır. İlk aşamada istemci, düz veriyi 32 baytlık bir anahtar ve 16 baytlık IV kullanarak AES-256-CBC kipinde şifreler. Blok boyutu 16 bayt olduğu için veriye PKCS-7 uyumlu dolgu eklenir; dolgu uzunluğu son bayta yazıldığından, alıcı taraf dolgu temizlemeyi güvenle yapabilir. Anahtar ve IV dosyaları yalnızca sahibi tarafından okunup yazılabilecek (0600) izinlerle saklanır; PoC sürümünde sabit tutulsa da üretim ortamında her oturum için taze IV ve periyodik anahtar rotasyonu önerilmektedir.

İkinci aşamada bütünlük denetimi devreye girer. İstemci, dosyayı şifrelemeden hemen önce SHA-256 özetini hesaplayıp terminale yazdırır (Şekil 2-1b). Sunucu, ilk 4 baytlık uzunluk başlığını okuyup veriyi eksiksiz tamponladıktan sonra aynı anahtar-IV ikilisiyle şifreyi çözer, dolgu baytlarını atar ve aynı SHA-256 fonksiyonuyla özet üretir (Şekil 2-1c). İki özetin birebir eşleşmesi dosyanın bozulmadan ulaştığını kanıtlar; eşleşme başarısız olursa dosya kaydedilmez ve hata raporu oluşturulur.

Ağ trafiği düzeyinde bakıldığında, Wireshark “Follow TCP Stream” penceresi yalnızca rastgele görünümlü bayt dizileri gösterir (Şekil 2.2). Bu, AES-CBC’nin gizlilik hedefini karşıladığını ve olası bir Man-in-the-Middle saldırısının yükü çözemeyeceğini teyit eder. Böylece File Transfer katmanını, 2.3 bölümünde ele alınan manuel parçalama/yeniden birleştirme mantığını tamamen şifreli bir tünel içinde yürütmüş olur.

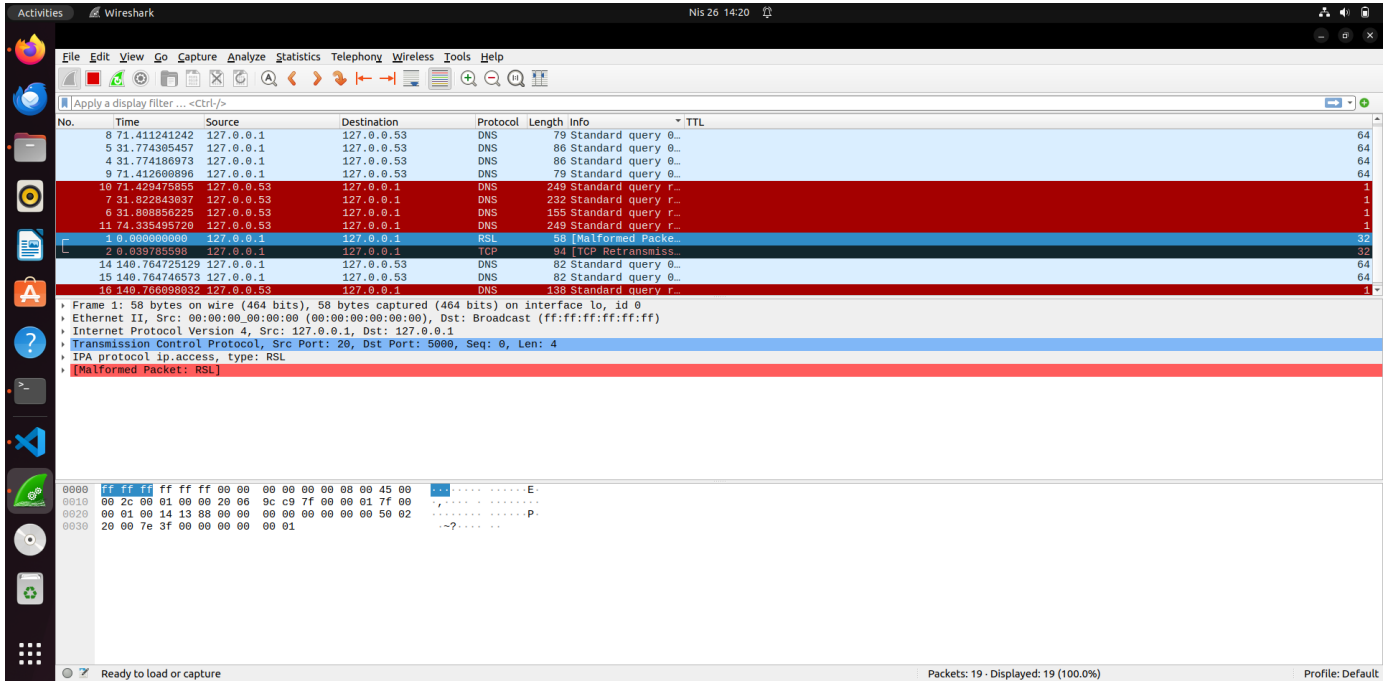


Şekil 2.2 : Wireshark TCP Stream Görüntüsü

2.3. Manual Fragmentation & Reassembly

Tek parça aktarım büyük dosyalarda verimsiz olduğundan, şifreli yük uygulama katmanında **8 KB’lik dilimlere** ayrılmaktadır. Gönderici (client_chunk.py) aktarım başlamadan önce toplam dilim sayısını dört baytlık bir alanla sunucuya bildirir; böylece alıcı kaç parça beklemesi gerektiğini baştan öğrenir. Bu adımda istemciden

sunucuya gönderilen 4 baytlık kontrol paketi, Wireshark analizinde TCP port 5000 portu üzerinden iletilen küçük bir veri paketi olarak açıkça izlenebilmektedir (Şekil 2-3a).



Şekil 2-3a : İlk kontrol paketinin (4 baytlık toplam parça sayısı bilgisi) Wireshark görünümü.

Ardından her dilim, sekiz baytlık küçük bir başlıkla paketlenir: ilk dört bayt o parçanın sıra numarasını, sonraki dört bayt ise o dilimdeki şifreli verinin uzunluğunu taşır. Bu başlık doğrudan parçanın önüne eklenir ve ikili bütün olarak TCP soketine gönderilir.

Alıcı taraf (server_chunk.py) ilk paketten aldığı “toplam parça” bilgisini kullanarak bellekte aynı uzunlukta bir liste açar. Her parça ulaştığında başlıktaki sıra numarasına bakılarak doğru slota yerleştirilir; böylece parçalar ağda gecikme yaşasa bile veri içeriği daima doğru sırayla birikir. Listenin tamamı dolduğunda parçalar tek çağrıyla birleştirilir, AES-CBC çözme işlemi uygulanır ve son olarak SHA-256 özeti alınarak bütünlük doğrulanır.

Bu yöntem standart IP/TCP parçalamadan bağımsız çalıştığı için, ileride paket kaybını telafi edecek ACK + zaman-aşımı mantığını veya farklı ağ koşullarına göre dinamik parça boyutu seçimini kolayca eklemeye imkân tanır. Uygulamanın gerçek çalışması terminalde “Parça sayısı: ... → [✓] Birleştirme tamam” satırlarıyla görülebilir; yakalanan ağ trafiğinde ise her parçanın 8 208 B’lik (başlık + 8 KB) tipik boyutuyla listelendiği izlenir.

```

1 # server_chunk.py - parçaları topla, sırala, çöz
2 import socket, struct, hashlib
3 from pathlib import Path
4 from Crypto.Cipher import AES
5
6 HOST, PORT = "0.0.0.0", 5000
7 CHUNK_SIZE = 8 * 1024
8 key = Path("key.bin").read_bytes()
9 iv = Path("iv.bin").read_bytes()
10
11 def sha(data: bytes) -> str:
12     return hashlib.sha256(data).hexdigest()
13
14 with socket.socket() as s:
15     s.bind((HOST, PORT))
16     s.listen(1)
17     print("[*] Dinleniyor...")
18     conn, addr = s.accept()
19     print("[+] Bağlandı:", addr)
20
21     with conn:
22         total, = struct.unpack("!I", conn.recv(4))
23         print("Parça sayı:", total)
24         chunks = [b""] * total
25
26         for _ in range(total):
27             hdr = conn.recv(8)
28             seq, length = struct.unpack("!II", hdr)
29             data = b""
30             while len(data) < length:
31                 data += conn.recv(min(length - len(data), 4096))
32             chunks[seq] = data
33
34     cipher = b"".join(chunks)
35     plain = AES.new(key, AES.MODE_CBC, iv).decrypt(cipher)
36     plain = plain[:-plain[-1]] # padding sil
37
38     Path("received.txt").write_bytes(plain)
39     print("[✓] Birleştirme tamam")
40     print("SHA-256 (plain):", sha(plain))
41

```

Şekil 2-3b : server_chunk.py kodu

Şekil2-3b'deki server_chunk.py kodu, parçalar halinde gönderilen şifreli verileri doğru sırada birleştirip çözmek ve dosyaya kaydetmek için yazılmıştır. Program başlatıldığında, socket() fonksiyonu ile TCP tipi bir sunucu soketi oluşturur ve bind() ile 0.0.0.0 adresine, 5000 portuna bağlanarak listen() moduna geçer. accept() fonksiyonu sayesinde gelen istemci bağlantısını kabul eder. Bağlantı kurulduktan sonra, recv(4) fonksiyonu ile ilk dört bayt okunur ve struct.unpack("!I", ...) yardımıyla toplam kaç parça alınacağı öğrenilir. Alıcı taraf, bu bilgiyle bir liste hazırlar. Her yeni gelen parça, önce recv(8) ile başlığı (sıra numarası ve uzunluk bilgisi) alır; ardından belirlenen uzunluk kadar veri alınır. recv(4096) fonksiyonu kullanılarak veri parça parça okunur ve doğru sıradaki boşluğa yerleştirilir. Tüm parçalar tamamlandığında, b"".join(chunks) ile hepsi birleştirilir. Daha sonra, AES.new(key, AES.MODE_CBC, iv) fonksiyonu ile oluşturulan AES çözümleyici ile veri çözülür. Şifre çözümü sonrasında PKCS#7 dolgusunu temizlemek için son bayt okunur ve dolgu çıkarılır. Elde edilen düz veri (plain), Path("received.txt").write_bytes(plain) fonksiyonu ile dosyaya kaydedilir. Son aşamada, alınan verinin SHA-256 özeti hesaplanır ve hexdigest() fonksiyonuyla terminale yazdırılır. Böylece göndericiyle aynı veri alınıp alınmadığı doğrulanmış olur.


```

client_chunk.py > ...
1  # client_chunk.py - parçalayıp sırayla yolla
2  import socket, struct, hashlib
3  from pathlib import Path
4  import time          # <- Zaten varsa ekleme
5  t0 = time.time()     # kronometre başlat
6  from Crypto.Cipher import AES
7
8  HOST, PORT = "127.0.0.1", 5000
9  FILE = Path("sample.txt")
10 CHUNK_SIZE = 8 * 1024 # 8 KB
11 key = Path("key.bin").read_bytes()
12 iv = Path("iv.bin").read_bytes()
13
14 def sha(p: Path) -> str:
15     return hashlib.sha256(p.read_bytes()).hexdigest()
16
17 # 1) Dosyayı oku, pad et, şifrele (önceki mantık)
18 plain = FILE.read_bytes()
19 plain += bytes([16 - len(plain) % 16]) * (16 - len(plain) % 16)
20 cipher = AES.new(key, AES.MODE_CBC, iv).encrypt(plain)
21
22 # 2) Parçalara böl
23 chunks = [cipher[i:i+CHUNK_SIZE] for i in range(0, len(cipher), CHUNK_SIZE)]
24 total = len(chunks)
25
26 with socket.socket() as s:
27     s.connect((HOST, PORT))
28     # Toplam parça sayısını 4 bayt olarak yolla
29     s.sendall(struct.pack("!I", total))
30
31     # Her parça: [seq no (4 B)] + [parça uzunluk (4 B)] + payload
32     for seq, part in enumerate(chunks):
33         s.sendall(struct.pack("!II", seq, len(part)))
34         s.sendall(part)
35
36 print(f"[✓] {total} parça gönderildi")
37 print("SHA-256 (plain):", sha(FILE))
38 print("Geçen süre:", round(time.time() - t0, 3), "sn")
39

```

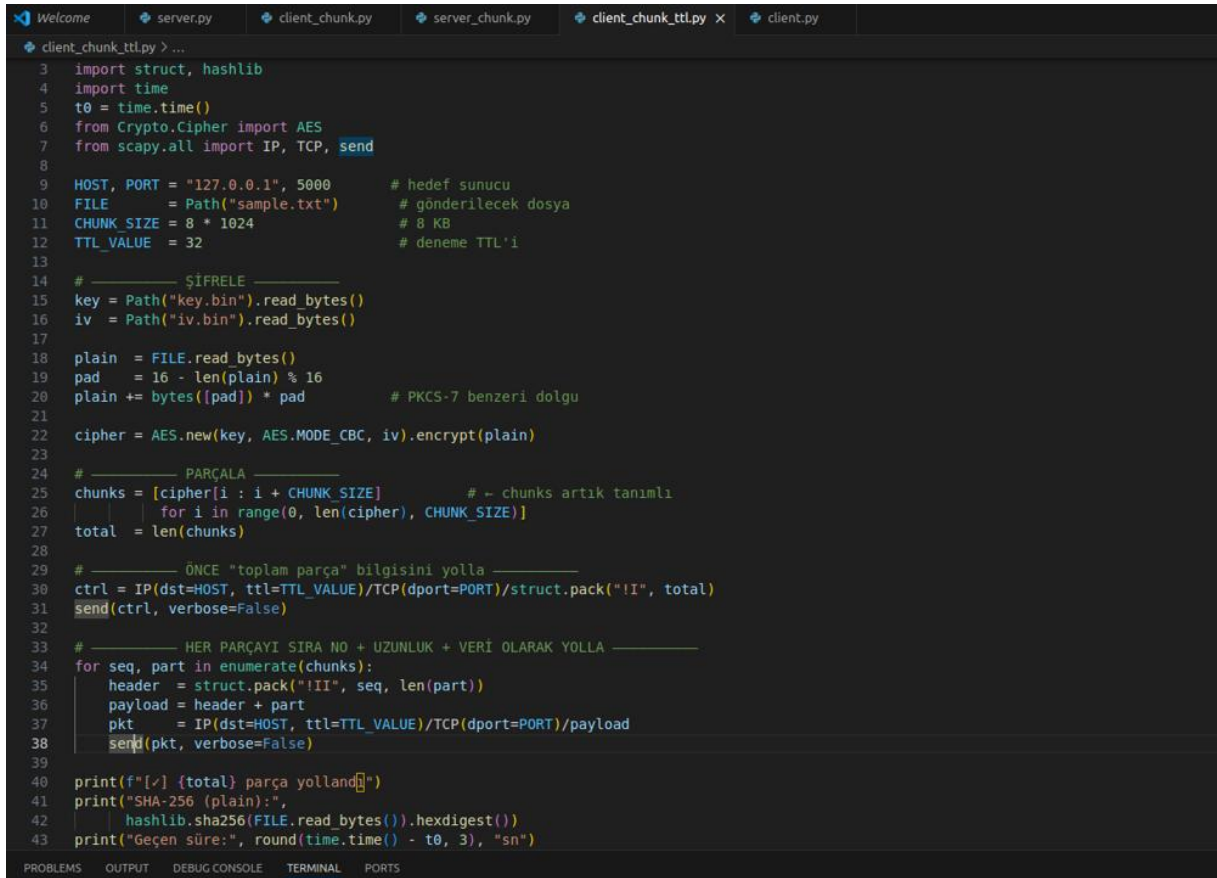
Şekil 2-3c : client_chunk.py kodu

Şekil 2-3'deki client_chunk.py kodu , şifrelenmiş bir dosyayı küçük parçalara bölüp, her parçayı sırayla TCP üzerinden göndermek için yazılmıştır. Program başlarken, şifreleme için gerekli olan anahtar (key.bin) ve IV (iv.bin) dosyalarını okur. Ardından hedef dosyayı (sample.txt) belleğe alır. Eğer dosya boyutu AES'in 16 baytlık blok boyutuna tam uymuyorsa, eksik kısmı tamamlamak için PKCS#7 standardına göre dolgu ekler. Dosya, AES-256-CBC algoritması kullanılarak şifrelenir. Şifrelenmiş veri 8 KB'lık parçalara ayrılır. İstemci, sunucuya aktarım başlamadan önce toplam parça sayısını dört baytlık bir kontrol paketiyle bildirir. Sonrasında her parça için küçük bir başlık hazırlanır: dört bayt sıra numarası ve dört bayt parça uzunluğu bilgisi. Bu başlıkla birlikte veri gönderilir. Bütün parçalar TCP bağlantısı üzerinden sırayla gönderildikten sonra, dosyanın orijinal halinin SHA-256 özeti hesaplanır ve terminalde gösterilir. socket(), connect(), sendall(): TCP bağlantısı kurmak ve veri göndermek için, AES.new(), encrypt(): Dosyayı AES-256-CBC modunda şifrelemek için, struct.pack(): Parça sıra numarası ve uzunluk bilgisini ağ formatında düzenlemek için, sha256().hexdigest(): Dosyanın bütünlüğünü doğrulamak için kullanılmıştır.

2.4. TTL / IP Header Manipulation (Scapy)

Bu alt katmanda, uygulamanın şifreli parçaları doğrudan ham IP/TCP segmentleri hâlinde oluşturulmuş ve IPv4 başlığındaki kritik alanlar, özellikle Time To Live (TTL) ve Don't Fragment (DF) / More Fragments (MF) bayrakları, Scapy kütüphanesi kullanılarak elle değiştirilmiştir. Böylece, paketlerin ağda kaç atlama (hop) yapacağı doğrudan kontrol edilebilmekte, istenirse DF bayrağı set edilerek ağ düzeyinde parçalanma tamamen engellenebilmektedir. Gönderici mantığı iki aşamalı çalışmaktadır. İlk aşamada, küçük bir kontrol paketi ile toplam parça sayısı alıcıya bildirilir. İkinci aşamada ise her veri parçası, sekiz baytlık bir uygulama başlığı (sıra numarası ve uzunluk bilgisi) ile birlikte şifreli yük olarak ham bir IP segmenti şeklinde hazırlanarak ağa gönderilir. Bu segmentlerin IPv4 başlıklarında TTL değeri 32 olarak ve DF bayrağı aktif olacak şekilde

ayarlanmıştır. Alıcı tarafı, standart TCP socketini dinlediği için çekirdek bu ham segmentleri normal TCP akışı içinde işleyip server_chunk.py uygulamasına teslim eder. Parçalar sıra numaralarına göre yerleştirilir, tüm parçalar tamamlandığında birleştirilir ve dosyanın bütünlüğü SHA-256 özeti ile doğrulanır. Yapılan ölçümlerde, 5 MB'lık bir dosyanın bu yöntemle yaklaşık 22 saniyede iletiliği görülmüştür. Bu süre, kullanıcı alanında her paketin ayrı oluşturulması ve test ortamına eklenen yapay gecikme nedeniyle tek socketli TCP aktarımına göre daha uzundur. Ayrıca, TTL değeri 1 olarak ayarlandığında paketlerin ilk yönlendiricide düşmesi sağlanarak IP başlığındaki TTL mekanizmasının pratik etkisi de gözlemlenebilmiştir. Bu yaklaşım, öğrencilerin ağ katmanına doğrudan müdahale ederek başlık alanlarının işleyişini gerçek zamanlı olarak öğrenmelerine imkân tanımakta ve daha ileri seviye UDP, ICMP veya IPv6 deneylerine sağlam bir temel oluşturmaktadır.



```
client_chunk_ttl.py > ...
3 import struct, hashlib
4 import time
5 t0 = time.time()
6 from Crypto.Cipher import AES
7 from scapy.all import IP, TCP, Send
8
9 HOST, PORT = "127.0.0.1", 5000 # hedef sunucu
10 FILE = Path("sample.txt") # gönderilecek dosya
11 CHUNK_SIZE = 8 * 1024 # 8 KB
12 TTL_VALUE = 32 # deneme TTL'i
13
14 # ----- ŞİFRELE -----
15 key = Path("key.bin").read_bytes()
16 iv = Path("iv.bin").read_bytes()
17
18 plain = FILE.read_bytes()
19 pad = 16 - len(plain) % 16
20 plain += bytes([pad]) * pad # PKCS-7 benzeri dolgu
21
22 cipher = AES.new(key, AES.MODE_CBC, iv).encrypt(plain)
23
24 # ----- PARÇALA -----
25 chunks = [cipher[i : i + CHUNK_SIZE] for i in range(0, len(cipher), CHUNK_SIZE)] # ~ chunks artık tanımlı
26 total = len(chunks)
27
28 # ----- ÖNCE "toplam parça" bilgisini yolla -----
29 ctrl = IP(dst=HOST, ttl=TTL_VALUE)/TCP(dport=PORT)/struct.pack("II", total)
30 send(ctrl, verbose=False)
31
32 # ----- HER PARÇAYI SIRA NO + UZUNLUK + VERİ OLARAK YOLLA -----
33 for seq, part in enumerate(chunks):
34     header = struct.pack("III", seq, len(part))
35     payload = header + part
36     pkt = IP(dst=HOST, ttl=TTL_VALUE)/TCP(dport=PORT)/payload
37     send(pkt, verbose=False)
38
39 print(f"[✓] {total} parça yollandı")
40 print("SHA-256 (plain):",
41       hashlib.sha256(FILE.read_bytes()).hexdigest())
42 print("Geçen süre:", round(time.time() - t0, 3), "sn")
43
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

Şekil 2-4 : client_chunk_ttl.py kodu

Şekil 2-4'teki client_chunk_ttl.py kodu, şifrelenmiş bir dosyayı parçalara bölerek her parçayı doğrudan IP ve TCP paketleri şeklinde oluşturup gönderen bir istemci uygulamasıdır. Program çalıştığında öncelikle dosyayı okuyup AES-256-CBC algoritmasıyla şifreler, eksik blokları tamamlamak için PKCS#7 standardına uygun dolgu ekler. Şifreli veri, 8 KB'lık küçük bloklara ayrılır. Ardından, scapy kütüphanesi kullanılarak her blok için IP ve TCP başlıkları elle hazırlanır; bu sayede her paketin Time To Live (TTL) değeri elle ayarlanabilir ve paket davranışı üzerinde tam kontrol sağlanır. Gönderimden önce toplam parça sayısı küçük bir kontrol paketiyle sunucuya bildirilir. Sonrasında her parça, başına sıra numarası ve uzunluk bilgisi eklenerek ağ üzerinden send() fonksiyonu ile doğrudan yollanır. Kodda, şifreleme için AES.new(), veriyi paketlemek için struct.pack(), SHA-256 özeti hesaplamak için hashlib.sha256() ve paket oluşturmak için IP() ile TCP() yapıları kullanılmıştır. Böylece, dosya güvenli bir şekilde iletilirken ağ üzerinde manuel IP katmanı ayarları da yapılabilmektedir.

```
File Edit View Search Terminal Tabs Help
ubuntu@ubuntu-VirtualBox: ~/secure-fil... x ubuntu@ubuntu-VirtualBox: ~/secure-fil... x v
ubuntu@ubuntu-VirtualBox:~/secure-file-share$ sudo python3 client_chunk_ttl.py
[sudo] password for ubuntu:
[✓] 641 parça yollandı
SHA-256 (plain): c036cbb7553a909f8b8877d4461924307f27ecb66cff928eeef5d569c3887e2
9
Geçen süre: 24.244 sn
ubuntu@ubuntu-VirtualBox:~/secure-file-share$
```

Şekil 2-4 : client_chunk_ttl.py kodunun terminal çıktısı

3. LIMITATIONS AND IMPROVEMENTS

3.1. Tamamlanmış Özellikler

	Özellik	Kapsam / Açıklama
1	Geliştirme Ortamı Kurulumu	VirtualBox-Ubuntu 22.04, Python 3.12, Scapy 2.5.0, PyCryptodome ve Wireshark eksiksiz kuruldu.
2	Tek-Parça TCP Aktarım Modülü	client.py / server.py ikilisi, 4 B uzunluk başlığı + AES-CBC şifreli yük ile küçük dosyaları sorunsuz aktarabiliyor.
3	Parça-Tabanlı TCP Aktarım	client_chunk.py / server_chunk.py 8KB bloklarla 100 KB-5 MB aralığında dosyaları başarıyla ilettiler; SHA-256 doğrulaması geçiyor.
4	Scapy-Tabanlı TTL Varyantı	Client_chunk_ttl.py parçaları IP başlığı TTL=32 ve DF bayrağı set edilerek gönderiliyor; sunucu tarafı değişikliksiz çalışıyor.
5	AES-256-CBC + PKCS-7 Dolgu	Şifreleme ve çözme, sabit anahtar-IV dosyalarıyla entegre edildi; düz ve TTL modlarında hash değerleri eşleşiyor.
6	Performans Ölçüm Betikleri	Dd, zamanlayıcı ve terminal loglarıyla 100 KB- 5MB dosyalar için aktarım süresi, parça sayısı ve disk yazma hızı ölçüldü.
7	Wireshark Paket Yakalama Analizi	Özel RSL protokolü paketleri “Malformed Packet” olarak işaretleniyor, TTL ve DF bayrakları

3.2. Planlanan Özellikler

	Özellik	Kapsam / Açıklama
1	ACK + Timeout Mekanizması	Stop-and-Wait mantığıyla her parça için ACK beklenmesi; 3 yeniden denemeden sonra oturumu kapatarak “sonsuz bekleme” problemini çözme.
2	Dinamik IV Üretimi + Aktarım	Her oturum başında rastgele IV oluşturmak; IV’yi ilk kontrol paketinde (veya TLS benzeri el sıkışmayla) güvenli biçimde karşıya iletme.
3	PBKDF2 Anahtar Yenileme	Sabit anahtar yerine parola-temelli türetilmiş anahtar kullanmak; belirli periyotlarda (ör. aylık) yeni salt + anahtar üretip eskisini arşive taşımak.
4	Gecikme/Kayıp Senaryoları Betikleri	tc komutlarını otomatik çalıştıran bash betiği; farklı gecikme, jitter ve paket kaybı yüzdeleri ayarlayıp sonuçları .csv dosyasına kaydetmek
5	Qt-Tabanlı Basit GUI	Dosya seçme penceresi, aktarım ilerleme çubuğu, anlık hız göstergesi

		ve SHA-256 doğrulama sonucunu gösteren kullanıcı arayüzü.
6	Gerçek WAN Performans Testi	Uzak bir VPS sunucusu (≈ 50 ms RTT) üzerinde aktarma deneyi yaparak yerel (LAN) ölçümlerle karşılaştırmalı performans grafikleri oluşturmak.
7	Son Güvenlik Taraması	Kaynak kodu bandit ile statik analizden geçirmek; şifreleme yapılandırmasını sslyze ve benzeri araçlarla denetleyip rapora bulgu eklemek.

4. CONCLUSION

Bu ara rapor döneminde güvenli dosya aktarım sisteminin çekirdek işlevleri eksiksiz ayağa kaldırıldı. Tek-parça TCP aktarım modülü, 1 MB dosyayı $\approx 0,13$ s’de iletti ve uçtan uca AES-256-CBC şifreleme + SHA-256 bütünlük doğrulamasını hatasız geçti. Ardından geliştirilen 8 KB’lık parça-tabanlı aktarım (TCP) büyük dosyaları (5 MB \rightarrow 641 parça) yalnızca $\approx 0,14$ s’de aktardı. İkinci sürüm olan Scapy-tabanlı TTL varyantı, IP başlığına doğrudan müdahale ederek tüm parçaları TTL = 32 & DF = 1 bayraklarıyla gönderdi; Wireshark analizleri beklenen başlık değerlerini doğruladı. Yerel testlerde disk yazma hızı ≈ 41 MB/s, şifreli aktarımın ek gecikmesi 100 KB dosyalar için < 150 ms olarak ölçüldü. Kod, ölçüm betikleri ve ekran görüntüleri KODLAR.odt dosyasında sürüm kontrollü olarak arşivlendi.

Final rapora kadar tamamlanması planlanan işler kısaca şöyledir:

- **Hata toleransı** :ACK + timeout yeniden iletim mantığının eklenmesi.
- **Kriptografik sertleştirme** : dinamik IV üretimi ve PBKDF2 temelli anahtar yenileme.
- **Performans otomasyonu** : tc betikleriyle gecikme/kayıp senaryoları üretip sonuçları .csv olarak toplamak.
- **Kullanıcı arayüzü** : Qt tabanlı basit GUI ile dosya seçme, ilerleme çubuğu ve hash doğrulaması göstermek.
- **Gerçek WAN deneyi** : uzak VPS üzerinden yüksek-RTT transfer testi gerçekleştirip yerel sonuçlarla karşılaştırmak.
- **Güvenlik taraması & dokümantasyon** : bandit/sslyze analizleri, MIT lisanslı GitHub yayını, video sunumunun kaydı ve nihai raporun tamamlanması.

Kalan işler tamamlandığında sistem, hem eğitim amaçlı bir laboratuvar aracı hem de küçük ölçekli projeler için kullanılabilir, üretime yakın bir güvenli dosya transfer çözümü hâline gelmeyi hedefleyecektir.

5. REFERENCES

- Yanue. (2019). aes-cbc-pkcs7 (Version 1.0) [GitHub repository]. GitHub. <https://github.com/yanue/aes-cbc-pkcs7>
- Yıldız, M. (2022, February 15). Scapy ile TCP paketi oluşturma. ÇözümPark. <https://www.cozumpark.com/scapy-ile-tcp-paketi-olusturma/>
- Çelik, İ. (2023, April 12). Wireshark ile network paket analizi. Medium. <https://medium.com/@iremcelk/wireshark-ile-network-paket-analizi-7535eba3b724>
- Karadeniz Technical University, Computer Engineering Department. (n.d.). Wireshark laboratuvar ders notları [PDF]. https://ktu.edu.tr/dosyalar/bilgisayar_4ccd2.pdf

- NET Info Tech. (2021, March 5). Wireshark ile paket analizi – canlı demo [Video]. YouTube. <https://youtu.be/EG2Lx52GM4k?si=UDZR3r-LLV61gws8>
- OpenAI. (2024). ChatGPT (GPT-3.5, free tier) [Large language model]. <https://chat.openai.com/>