GAZİ ÜNİVERSİTESİ
TEKNOLOJİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

BMT-440 MOBİL GÜVENLİK PROJE ÖDEVİ
PROJE RAPORU
2025-2026 GÜZ YARIYILI

AD SOYAD : CEYDA EMRE
NUMARA: 201816707
DERS HOCASI: ÖMER KİRAZ

1. Özet (Abstract)

Android işletim sistemi, mobil cihaz pazarında en yaygın kullanılan platformlardan biri olup, açık ekosistemi nedeniyle zararlı yazılımlar için önemli bir hedef haline gelmiştir. Bu çalışmada, Android uygulama paketlerinin (APK) **statik analiz** yöntemleriyle incelenmesi ve elde edilen özellikler üzerinden **makine öğrenmesi** algoritmaları ile zararlı (malware) ve zararsız (benign) uygulamaların sınıflandırılması amaçlanmıştır.

Veri seti, AndroZoo veri tabanından elde edilen **500 APK**'dan oluşmakta olup sınıflar dengeli olacak şekilde **250 benign** ve **250 malware** içermektedir. APK'lar üzerinde Python ortamında **androguard** kütüphanesi kullanılarak; izin bilgileri, Dalvik opcode frekansları, kritik API çağrıları, string tabanlı anomaliler, paket adı entropisi, native kütüphane (.so) varlığı ve APK boyutu gibi **60'tan fazla statik özellik** çıkarılmıştır.

Bu özellikler kullanılarak **RandomForest** ve **XGBoost** sınıflandırıcıları eğitilmiş, %80 eğitim – %20 test ayrımı ve 10 katlı çapraz doğrulama (10-fold cross-validation) ile değerlendirme yapılmıştır. RandomForest modeli, test kümesi üzerinde **%88 doğruluk (accuracy)**, zararlı sınıfı için **0.91 precision**, **0.84 recall**, **0.88 F1 skoru** ve **0.952 AUC** (ROC eğrisi altında kalan alan) değerlerine ulaşmıştır. XGBoost modeli de benzer performans göstermiştir.

Özellik önem düzeyi analizi, özellikle **reflection temelli API çağrılarının** (getDeclaredMethod, invoke) ve çağrı ile ilişkili opcode'ların zararlı yazılımların tespitinde kritik rol oynadığını göstermiştir. Korelasyon analizi ise izin tabanlı özellikler, opcode frekansları ve API çağrıları arasında anlamlı ilişkiler bulunduğunu ortaya koymuştur. Sonuçlar, statik analiz ile makine öğrenmesi yöntemlerinin birlikte kullanılması halinde Android zararlı yazılımlarına karşı etkili ve ölçeklenebilir bir tespit mekanizması sağlayabileceğini göstermektedir.

2. Giriş (Introduction)

Android işletim sistemi; Google Play Store, üçüncü taraf mağazalar ve doğrudan APK kurulumu gibi pek çok dağıtım kanalına sahiptir. Bu durum, kullanıcıların kolayca uygulama yükleyebilmesini sağlarken, aynı zamanda kötü amaçlı yazılımlar için geniş bir saldırı yüzeyi oluşturur. **Android malware** türleri; SMS dolandırıcılığı, kimlik bilgisi hırsızlığı, kripto madenciliği, arka kapı (backdoor) oluşturma, ransomware gibi çok çeşitli zararlı amaçlar için kullanılmaktadır.

Zararlı yazılım tespiti için iki temel yaklaşım bulunmaktadır:

1. **Dinamik analiz**: Uygulamanın emülatör veya gerçek cihaz üzerinde, kontrollü bir ortamda çalıştırılması ve davranışlarının gözlemlenmesi.

2. **Statik analiz**: Uygulamanın çalıştırılmadan, APK içeriğinin (manifest, bytecode, kaynak dosyaları) incelenmesi.

Dinamik analiz davranış açısından çok zengin bilgi sağlasa da; zaman maliyeti yüksektir, büyük veri setleri için ölçeklenebilirliği sınırlıdır ve anti-analiz teknikleri ile kandırılabilir. Statik analiz ise **daha hızlı**, **daha ucuz** ve **büyük veri setlerinde uygulanabilir** bir yöntemdir. Dezavantajı ise obfuscation (kod karartma) tekniklerinden etkilenebilmesidir.

Bu çalışmada, tamamen statik analiz temelli bir yaklaşım benimsenmiş; APK'lardan elde edilen çok sayıda özellik, makine öğrenmesi yöntemleriyle birleştirilerek modern Android malware'lerinin tespiti hedeflenmiştir.

3. Materyal ve Yöntem (Materials and Methods)

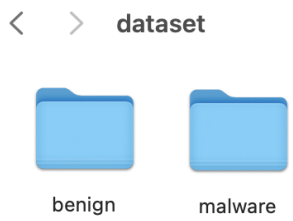
3.1 Veri Seti

Veri seti, akademik araştırmalarda sıkça kullanılan **AndroZoo** veri tabanından elde edilmiştir. AndroZoo; farklı kaynaklardan (Google Play, üçüncü taraf marketler vb.) toplanmış milyonlarca Android uygulamasını SHA-256 hash'leriyle birlikte barındıran bir koleksiyondur.

Bu çalışma kapsamında:

- **Toplam 500 APK** seçilmiştir.
- Sınıf dağılımı **dengeli** olacak şekilde:
    - 250 adet benign (zararsız)
    - 250 adet malware (zararlı)

APK dosyaları yerel ortamda:



şeklinde organize edilmiştir. Her APK, SHA-256 hash değerine göre AndroZoo API kullanılarak indirilmiş ve daha sonra kategorisine göre ilgili klasöre yerleştirilmiştir.

Dengeli veri kümesi kullanılması, **accuracy** ve diğer performans ölçütlerinin tek taraflı bir sınıf lehine sapmamasını sağlar. Bu sayede modelin her iki sınıfı da adil şekilde öğrendiğinden daha emin olunabilmektedir.

3.2 Kullanılan Araçlar

- **Python 3**: Genel geliştirme ortamı.
- **Androguard**: APK'lardan manifest, bytecode, string ve dosya sistemi bilgilerini çıkaran Python kütüphanesi.

- **pandas, numpy**: Veri setinin saklanması, işlenmesi ve teknoloji.
- **scikit-learn**: Makine öğrenmesi algoritmaları ve metrikler.
- **XGBoost**: Gradient boosting tabanlı güçlü bir sınıflandırıcı.
- **matplotlib, seaborn**: Görselleştirme (ROC eğrisi, feature importance, korelasyon haritası).

3.3 Özellik Çıkarımı (Feature Extraction)

Bu bölümde extract_features.py script'inin yaptığı işlemler detaylı biçimde anlatılmaktadır. Script'in amacı, her APK için belirli statik davranışları sayısal özelliklere dönüştürmek ve bunları satır satır bir CSV dosyasına yazmaktır.

extract_features.py:

```
import os

import re

import math

from typing import Optional, Dict

from androguard.misc import AnalyzeAPK

from tqdm import tqdm

import pandas as pd

# === PERMISSION SINIFLANDIRMALARI ===

DANGEROUS_PERMISSIONS = {

    "android.permission.READ_CONTACTS",

    "android.permission.WRITE_CONTACTS",

    "android.permission.GET_ACCOUNTS",

    "android.permission.READ_PHONE_STATE",

    "android.permission.CALL_PHONE",

    "android.permission.CAMERA",

    "android.permission.RECORD_AUDIO",

    "android.permission.ACCESS_FINE_LOCATION",
```

```
    "android.permission.ACCESS_COARSE_LOCATION",
    "android.permission.READ_SMS",
    "android.permission.SEND_SMS",
    "android.permission.RECEIVE_SMS",
    "android.permission.RECEIVE_MMS",
    "android.permission.READ_EXTERNAL_STORAGE",
    "android.permission.WRITE_EXTERNAL_STORAGE",
}
SIGNATURE_PERMISSIONS = {
    "android.permission.BIND_ACCESSIBILITY_SERVICE",
    "android.permission.BIND_AUTOFILL_SERVICE",
    "android.permission.BIND_VPN_SERVICE",
    "android.permission.BIND_DEVICE_ADMIN",
    "android.permission.BIND_NOTIFICATION_LISTENER_SERVICE",
}
# === OPCODE'LAR ===
IMPORTANT_OPCODES = {
    "invoke-virtual",
    "invoke-static",
    "const-string",
    "new-instance",
    "move",
    "goto",
}
# === API SIGNATURE'LAR ===
API_SIGNATURES = {
```

```
    "getDeviceId": "api_getDeviceId",
    "getSubscriberId": "api_getSubscriberId",
    "getLine1Number": "api_getLine1Number",
    "sendTextMessage": "api_sendTextMessage",
    "exec": "api_exec",
    "getRuntime": "api_runtime_getRuntime",
    "openConnection": "api_openConnection",
    "Socket": "api_newSocket",
    "Cipher.getInstance": "api_cipher_getInstance",
    "SecretKeySpec": "api_secretKeySpec_init",
    "Class.forName": "api_class_forName",
    "getDeclaredMethod": "api_getDeclaredMethod",
    "invoke": "api_method_invoke",
}
# === STRING ANALİZİ ===
def analyze_strings(a):
    strings = set()
    try:
        for s in a.get_strings():
            if isinstance(s, str):
                strings.add(s)
    except:
        pass
    url_count = sum(1 for x in strings if "http://" in x or "https://" in x)
    ip_count = sum(1 for x in strings if re.match(r"\d+\.\d+\.\d+\.\d+", x))
    base64_count = sum(1 for x in strings if re.match(r"[A-Za-z0-9+/]{20,}={0,2}$", x))
```

```
    hex_count = sum(1 for x in strings if re.match(r"^[0-9A-Fa-f]{20,}$", x))
    long_word_count = sum(1 for x in strings if len(x) > 40)
    return url_count, ip_count, base64_count, hex_count, long_word_count
# === PACKAGE NAME ENTROPY ===
def entropy(s: str) -> float:
    if not s:
        return 0.0
    prob = [float(s.count(c)) / len(s) for c in dict.fromkeys(s)]
    return -sum(p * math.log(p, 2) for p in prob)
# === TEK APK ÖZELLİK ÇIKARIMI ===
def extract_features_from_apk(apk_path: str) -> Optional[Dict]:
    try:
        a, d, dx = AnalyzeAPK(apk_path)
        permissions = a.get_permissions()
        dangerous_count = sum(1 for p in permissions if p in DANGEROUS_PERMISSIONS)
        signature_count = sum(1 for p in permissions if p in SIGNATURE_PERMISSIONS)
        activities = len(a.get_activities())
        services = len(a.get_services())
        receivers = len(a.get_receivers())
        # Opcode sayımı
        opcode_counts = {f"op_{op}": 0 for op in IMPORTANT_OPCODES}
        # API call sayımı
        api_counts = {v: 0 for v in API_SIGNATURES.values()}
        for method in dx.get_methods():
            if method.is_external():
                continue
```

```
    for block in method.get_basic_blocks():
        for ins in block.get_instructions():
            op = ins.get_name()
            if op in IMPORTANT_OPCODES:
                opcode_counts[f"op_{op}"] += 1
            # API çağrısı tespiti
            out = str(ins.get_output())
            for api_key, api_name in API_SIGNATURES.items():
                if api_key in out:
                    api_counts[api_name] += 1
# String analizi
url_c, ip_c, b64_c, hex_c, long_c = analyze_strings(a)
# Native library var mı?
has_native_lib = 1 if any(f.endswith(".so") for f in a.get_files()) else 0
# Package name entropy
pkg = a.get_package()
pkg_entropy = entropy(pkg)
# APK size
apk_size = os.path.getsize(apk_path)
features = {
    "permissions_count": len(permissions),
    "dangerous_permissions": dangerous_count,
    "signature_permissions": signature_count,
    "package_entropy": pkg_entropy,
    "has_native_lib": has_native_lib,
    "apk_size": apk_size,
```

```
            "activities": activities,
            "services": services,
            "receivers": receivers,
            "apk_name": os.path.basename(apk_path),
        }
        features.update(opcode_counts)
        features.update(api_counts)
        return features
    except Exception as e:
        print(f"[HATA] {apk_path}: {e}")
        return None
# === DATASET İŞLEME ===
def process_dataset(path, label):
    rows = []
    files = [f for f in os.listdir(path) if f.endswith(".apk")]
    for f in tqdm(files, desc=f"{label} analiz ediliyor"):
        apk_path = os.path.join(path, f)
        feats = extract_features_from_apk(apk_path)
        if feats:
            feats["class"] = label
            rows.append(feats)
    return rows
# === MAIN ===
if __name__ == "__main__":
    benign = process_dataset("dataset/benign", "benign")
    malware = process_dataset("dataset/malware", "malware")
```

```
df = pd.DataFrame(benign + malware)

df.to_csv("output/features.csv", index=False)

print("[OK] → output/features.csv oluşturuldu")
```

3.3.1 İzin Tabanlı Özellikler

Kodda tanımlanan:

```
DANGEROUS_PERMISSIONS = {...}
SIGNATURE_PERMISSIONS = {...}
```

kümeleri, Android'in **tehlikeli (dangerous)** ve **imza seviyesinde (signature)** izinlerini içerir. Her APK için:

- permissions_count: Toplam istenen izin sayısı
- dangerous_permissions: Dangerous izinlerin sayısı
- signature_permissions: Signature izinlerin sayısı

hesaplanır.

Neden önemli?

- Zararlı yazılımlar genellikle kullanıcı verilerine (SMS, rehber, konum) agresif erişim ister.
- Signature izinler normal uygulamalar tarafından çok nadir kullanılır; sistem veya üretici uygulamalarına özeldir.

Dolayısıyla izin profili, uygulamanın niyeti hakkında güçlü bir ipucu sunar.

3.3.2 Manifest ve Bileşen Sayıları

Androguard ile:

```
activities = len(a.get_activities())
services  = len(a.get_services())
receivers = len(a.get_receivers())
```

değerleri elde edilmiştir.

Bu bileşenler; uygulamanın ne kadar karmaşık olduğu, arka planda ne kadar servis veya broadcast receiver çalıştırdığı gibi ölçümler sunar. Zararlı yazılımlar çoğunlukla:

- Arka planda çalışan servisler
- Boot tamamlandığında tetiklenen receiver'lar

tanımlar; bu nedenle bileşen sayıları da bir davranış göstergesidir.

3.3.3 Opcode Frekansları

IMPORTANT_OPCODES kümesindeki opcode'ların frekansı sayılmıştır:

- invoke-virtual, invoke-static → Metot çağrıları; davranışın yoğunluğunu ve karmaşıklığı gösterir.
- const-string → Uygulama içinde metin, URL, IP gibi string'lerin ne kadar yoğun kullanıldığını gösterir.
- new-instance → Dinamik nesne oluşturma yoğunluğunu gösterir.
- move, goto → Register hareketleri ve kontrol akışı; obfuscation ile ilişkili olabilir.

Her bir opcode için:

```
opcode_counts[f"op_{op}"] += 1
```

şeklinde sayım yapılmış ve sonuçlar op_invoke-virtual, op_goto gibi kolonlara yazılmıştır.

3.3.4 API Çağrı İmzaları

API_SIGNATURES sözlüğü, belirli anahtar kelimeleri (örneğin "getDeviceId") ilgili özellik adlarıyla eşleştirir ("api_getDeviceId").

Kod:

```
out = str(ins.get_output())
for api_key, api_name in API_SIGNATURES.items():
    if api_key in out:
        api_counts[api_name] += 1
```

şeklinde her talimatın (instruction) metinsel çıktısını inceleyerek, belirli API çağrılarının kaç kez kullanıldığını sayar.

Bu API'ler, zararlı yazılımların tipik davranışlarını yansıtır:

- **getDeviceId, getSubscriberId, getLine1Number**: Cihaz kimliği ve kullanıcı hattı bilgisi toplama.

- **sendTextMessage**: Ücretli SMS gönderme.
- **exec, getRuntime**: Sistem komutları çalıştırma.
- **Socket, openConnection**: Ağ bağlantısı açma (C2 sunucularına bağlanma).
- **Cipher.getInstance, SecretKeySpec**: Şifreleme işlemleri ve gizli veri taşıma.
- **Class.forName, getDeclaredMethod, invoke**: Reflection (sınıf/metot isimlerini string olarak kullanıp çalışma zamanında çağırma).

Özellikle api_getDeclaredMethod ve api_method_invoke gibi reflection çağrıları, sonuçlarda en yüksek önem skoruna sahip olmuş ve malware tespitinde kritik rol oynamıştır.

3.3.5 String Analizi

analyze_strings(a) fonksiyonu, APK içindeki tüm string'leri gezerek:

- URL sayısı (string_url_count)
- IP adresi sayısı (string_ip_count)
- Base64 string sayısı (string_base64_count)
- Hex string sayısı (string_hex_count)
- 40 karakterden uzun string sayısı (string_long_word_count)

gibi ölçümleri çıkarır.

Bu string tipleri genellikle:

- Sunucu adresleri
- Şifrelenmiş veya gizlenmiş payload'lar
- Uzun şifreli token'lar

için kullanıldığı için zararlı yazılımlarda daha yoğun görünür.

3.3.6 Paket Entropisi

entropy(pkg) fonksiyonu, paket adının karakter dağılımından **Shannon entropisi** hesaplar. Entropi, bir dizgenin rastgelelik derecesini gösterir:

- Düşük entropi → Anlamlı ve okunabilir isim (örneğin: com.example.app)
- Yüksek entropi → Obfuscation sonucu rastgele karakterler (örneğin: a.b.c.a1b2c3)

Zararlı yazılımlar tespitten kaçmak için çoğu zaman paket adlarını rastgeleleştirir; bu da entropiyi yükseltir.

3.3.7 Native Kütüphane ve APK Boyutu

```
has_native_lib = 1 if any(f.endswith(".so") for f in a.get_files()) else 0
apk_size       = os.path.getsize(apk_path)
```

- has_native_lib: Uygulamanın .so uzantılı native kütüphane içerip içermediğini gösterir.
- apk_size: Dosya boyutu; daha ağır obfuscation ve ek modüller genellikle boyutu artırır.

3.4 Özelliklerin CSV'ye Yazılması

process_dataset(path, label) fonksiyonu, verilen klasördeki tüm .apk dosyalarını işleyerek her satırın sonuna class alanını ("benign" veya "malware") ekler. Tüm benign ve malware satırları birleştirilerek:

```
df.to_csv("output/features.csv", index=False)
```

komutuyla output/features.csv dosyası oluşturulur.

Bu dosya, makine öğrenmesi kısmında kullanılan **nihai özellik tablosudur**.

CSV'den Örnek Bir Satır:
permissions_count: 19
dangerous_permissions: 0
signature_permissions: 0
package_entropy: 3.12
has_native_lib: 1
apk_size: 1364280
activities: 14
services: 4
receivers: 4
op_invoke-virtual: 3119
op_invoke-static: 1275
op_const-string: 1271
op_new-instance: 637
op_move: 102
op_goto: 538
api_exec: 2
api_method_invoke: 12
api_getDeclaredMethod: 8

string_url_count: 9
string_ip_count: 346
string_base64_count: 0
string_hex_count: 80
class: malware
apk_name: “Omigo.apk”

3.5 Model Eğitimi ve Değerlendirme (train_ml.py)

train_ml.py betiği, özellikleri okur, veri setini böler, modelleri eğitir ve sonuçları hesaplar.

train_ml.py:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split, cross_val_score, KFold

from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

from sklearn.ensemble import RandomForestClassifier

from xgboost import XGBClassifier

# Grafik ayarları

plt.rcParams["figure.figsize"] = (10, 6)

# === VERİYİ YÜKLE ===

df = pd.read_csv("output/features.csv")

# sınıf etiketlerini sayısallaştır

X = df.drop(["class", "apk_name"], axis=1)

y = df["class"].map({"benign": 0, "malware": 1})  # binary mapping

# === 1) Train-Test Split ===

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.20, random_state=42, stratify=y

)
```

```
# === 2) RandomForest Eğitimi ===
rf = RandomForestClassifier(n_estimators=250, random_state=42)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
print("\n=== RandomForest Classification Report ===")
print(classification_report(y_test, y_pred))
print("\n=== RandomForest Confusion Matrix ===")
print(confusion_matrix(y_test, y_pred))
# === 3) 10-Fold Cross Validation ===
print("\n=== 10-Fold Cross Validation (RandomForest) ===")
kfold = KFold(n_splits=10, shuffle=True, random_state=42)
cv_scores = cross_val_score(rf, X, y, cv=kfold, scoring="accuracy")
print("Scores:", cv_scores)
print("Mean Accuracy:", cv_scores.mean())
print("Std:", cv_scores.std())
# === 4) ROC Curve + AUC ===
y_prob = rf.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.3f}")
plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — RandomForest")
```

```
plt.legend()
plt.savefig("output/roc_curve_rf.png")
plt.close()
# === 5) Feature Importance (Top 20) ===
importances = rf.feature_importances_
indices = np.argsort(importances)[-20:]  # en önemli 20 özellik
plt.figure(figsize=(10, 8))
plt.barh(range(len(indices)), importances[indices], align="center")
plt.yticks(range(len(indices)), [X.columns[i] for i in indices])
plt.xlabel("Importance")
plt.title("Feature Importance (Top 20 Features)")
plt.savefig("output/feature_importance.png")
plt.close()
# === 6) XGBoost Karşılaştırması ===
xgb = XGBClassifier(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=6,
    subsample=0.8,
    colsample_bytree=0.8,
    eval_metric="logloss",
    random_state=42,
)
xgb.fit(X_train, y_train)
y_pred_xgb = xgb.predict(X_test)
```

```
print("\n=== XGBoost Classification Report ===")

print(classification_report(y_test, y_pred_xgb))

print("\n=== XGBoost Confusion Matrix ===")

print(confusion_matrix(y_test, y_pred_xgb))

# === 7) Correlation Heatmap ===

plt.figure(figsize=(14, 10))

sns.heatmap(X.corr(), cmap="coolwarm", vmax=1.0, vmin=-1.0)

plt.title("Feature Correlation Heatmap")

plt.savefig("output/correlation_heatmap.png")

plt.close()

print("\n[OK] Tüm analizler tamamlandı. Çıktılar 'output' klasörüne kaydedildi.\n")
```

3.5.1 Train–Test Ayrımı

```
X = df.drop(["class", "apk_name"], axis=1)
y = df["class"].map({"benign": 0, "malware": 1})
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42, stratify=y
)
```

- **X**: Tüm sayısal özellikler
- **y**: Sınıf etiketleri (0 = benign, 1 = malware)
- **test_size = 0.20**: Verinin %20'si test için ayrılır (100 APK), %80'i eğitim için kullanılır (400 APK).
- **stratify=y**: Hem train hem test kümesinde sınıf dağılımının 50/50 kalmasını sağlar.
- **random_state = 42**: Aynı böllmenin tekrar üretilebilir olmasını sağlar.

3.5.2 RandomForest Eğitimi

```
rf = RandomForestClassifier(n_estimators=250, random_state=42)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
```

- **n_estimators=250**: 250 adet karar ağacı kullanılır; bu sayının artırılması çoğu zaman daha iyi genelleme sağlar.
- **RandomForest**, her bir ağacı verinin farklı bir alt kümesiyle eğiterek overfitting riskini düşüren bir ansambl (ensemble) yöntemidir.

3.5.3 XGBoost Eğitimi

```
xgb = XGBoostClassifier(...)
xgb.fit(X_train, y_train)
y_pred_xgb = xgb.predict(X_test)
```

XGBoost, gradient boosting tabanlı daha karmaşık bir modeldir; bu çalışmada RandomForest ile kıyaslama yapmak için kullanılmıştır.

3.5.4 10 Katlı Çapraz Doğrulama (10-Fold Cross Validation)

```
kfold = KFold(n_splits=10, shuffle=True, random_state=42)
cv_scores = cross_val_score(rf, X, y, cv=kfold, scoring="accuracy")
```

- Veri seti 10 eşit parçaya bölünür.
- Her seferinde 9 parça eğitim, 1 parça test olarak kullanılır.
- Bu işlem 10 kez tekrarlanır, her parça bir kez test kümesi olur.
- Sonuçlar cv_scores dizisinde saklanır; bu değerlerin ortalaması ve standart sapması modelin **genellenebilirlik** derecesini gösterir.

4. Deneysel Sonuçların Derinlemesine Analizi (Experimental Results)

Bu bölümde, elde edilen metrikler tek tek tanımlanacak, matematiksel olarak formüle edilecek ve siber güvenlik bağlamında yorumlanacaktır.

4.1 Karmaşıklık Matrisi (Confusion Matrix)

RandomForest modeli için elde edilen confusion matrix:

```
=== RandomForest Confusion Matrix ===
[[46  4]
 [ 8 42]]
```

Burada:

- **TP (True Positive)**: Malware olup doğru şekilde malware tahmin edilen sayısı = 42
- **TN (True Negative)**: Benign olup doğru şekilde benign tahmin edilen sayısı = 46
- **FP (False Positive)**: Benign olup yanlışlıkla malware tahmin edilen sayısı = 4
- **FN (False Negative)**: Malware olup yanlışlıkla benign tahmin edilen sayısı = 8

Toplam test kümesi büyüklüğü:

$$N = TP + TN + FP + FN = 42 + 46 + 4 + 8 = 100$$

Bu dört sayı, tüm diğer metriklerin temelini oluşturur.

Siber güvenlik yorumu:

- **FN (8)**: En riskli hatalar; çünkü gerçek malware'ler tespit edilmemiş olur.
- **FP (4)**: Temiz uygulamalar yanlış pozitif alarm üretir; bu da kullanıcı deneyimi açısından olumsuz ama güvenlik açısından FN kadar kritik değildir.
- Bu dağılım, modelin temelde dengeli bir hata profiline sahip olduğunu göstermektedir.

4.2 Doğruluk (Accuracy)

Tanım:
Modelin toplam tahminlerinin ne kadarını doğru yaptığıdır.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

RandomForest için:

$$Accuracy = \frac{42 + 46}{100} = 0.88$$

Bu, modelin test kümesindeki 100 APK'nın **%88'ini doğru sınıflandırdığı** anlamına gelir.

Avantajı:

- Genel performansı tek bir sayı ile özetler.
- Sınıflar dengeli olduğu için (50/50), bu çalışmada accuracy güvenilir bir ölçüdür.

Sınırlaması:

- Sınıflar dengesiz olduğunda (örneğin %95 benign, %5 malware) accuracy tek başına aldatıcı olabilir.
- Güvenlik uygulamalarında, özellikle **malware sınıfına ait recall ve precision** daha kritik olduğundan accuracy tek başına yeterli değildir.

4.3 Precision – "Alarm çaldığında gerçekten zararlı olma olasılığı"

Malware sınıfı için precision:

$$Precision_{malware} = \frac{TP}{TP + FP}$$

RandomForest için:

$$Precision_{malware} = \frac{42}{42 + 4} = \frac{42}{46} \approx 0.913$$

Yani:

> Model "bu APK malware" dediğinde, yaklaşık **%91** olasılıkla gerçekten malware'dir.

Bu, yanlış alarm (false positive) oranının düşük olduğunu gösterir.

Benign sınıfı için precision:

$$Precision_{benign} = \frac{TN}{TN + FN} = \frac{46}{46 + 8} \approx 0.85$$

Bu değer de modelin benign sınıfını da oldukça iyi ayırt ettiğini gösterir.

Siber güvenlik yorumu:

- Yüksek precision, sistemin gereksiz yere temiz uygulamaları kara listeye almamasını sağlar.
- Kullanıcı memnuniyeti için önemlidir; aksi halde antivirüs "her şeyi malware görüyor" algısı oluşur.

4.4 Recall (Sensitivity) – "Gerçek zararlıların ne kadarını yakaladık?"

Malware sınıfı için recall:

$$Recall_{malware} = \frac{TP}{TP + FN}$$

RandomForest için:

$$Recall_{malware} = \frac{42}{42 + 8} = \frac{42}{50} = 0.84$$

Yani:

Test kümesindeki zararlı APK'ların **%84'ü** doğru tespit edilmiştir.
Geriye kalan **%16'lık kısım (8 APK)** model tarafından gözden kaçırılmıştır.

Benign sınıfı için recall:

$$Recall_{benign} = \frac{TN}{TN + FP} = \frac{46}{46 + 4} = 0.92$$

Bu da temiz uygulamaların %92'sinin doğru tanındığını gösterir.

Siber güvenlik yorumu:

- Recall, **güvenlik açısından en kritik metriklerden biridir**.
- Çünkü FN sayısı arttıkça, sistem gerçek malware'leri kaçırmaya başlar.
- Bu çalışmada 0.84 değeri, pratik kullanıma uygun; ancak daha agresif sistemler için threshold ayarı değiştirilerek recall artırılabilir (precision bir miktar düşer).

4.5 Özgüllük (Specificity) ve Yanlış Pozitif Oranı (False Positive Rate)

Bu metrikler ROC eğrisi ile yakından ilişkilidir.

Specificity (özgüllük): Benign uygulamaların ne kadarının doğru benign tanındığını gösterir.

$$Specificity = \frac{TN}{TN + FP}$$

RandomForest için:

$$Specificity = \frac{46}{46 + 4} = 0.92$$

Yani temiz uygulamaların %92'si doğru sınıflanmıştır.

False Positive Rate (FPR) ise:

$$FPR = 1 - Specificity = \frac{FP}{FP + TN} = \frac{4}{50} = 0.08$$

Bu değer, ROC eğrisindeki yatay eksendeki (x-axis) değerin temelini oluşturur.

4.6 F1 Skoru – Precision ve Recall'un Dengesi

F1 skoru, precision ve recall'un harmonik ortalamasıdır.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Malware sınıfı için:

$$F1_{malware} \approx 2 \cdot \frac{0.91 \cdot 0.84}{0.91 + 0.84} \approx 0.88$$

F1 skoru, hem yanlış alarm (FP) hem de kaçan malware (FN) sayılarını aynı anda dikkate alır. Bu çalışmada her iki sınıf için de 0.88 civarında olması, modelin dengeli olduğunu göstermektedir.

4.7 10-Katlı Çapraz Doğrulama Sonuçları

cv_scores çıktısı:

[0.92, 0.92, 0.94, 0.82, 0.88, 0.86, 0.88, 0.80, 0.98, 0.8571]
Mean Accuracy: 0.8857
Std: 0.0523

Mean Accuracy (Ortalama Doğruluk) = 0.8857

- Veri seti hangi 9/1 kombinasyonuyla eğitilip test edilirse edilsin, model yaklaşık **%88–89** doğruluk civarında performans gösteriyor.

Standart Sapma (Std) = 0.0523

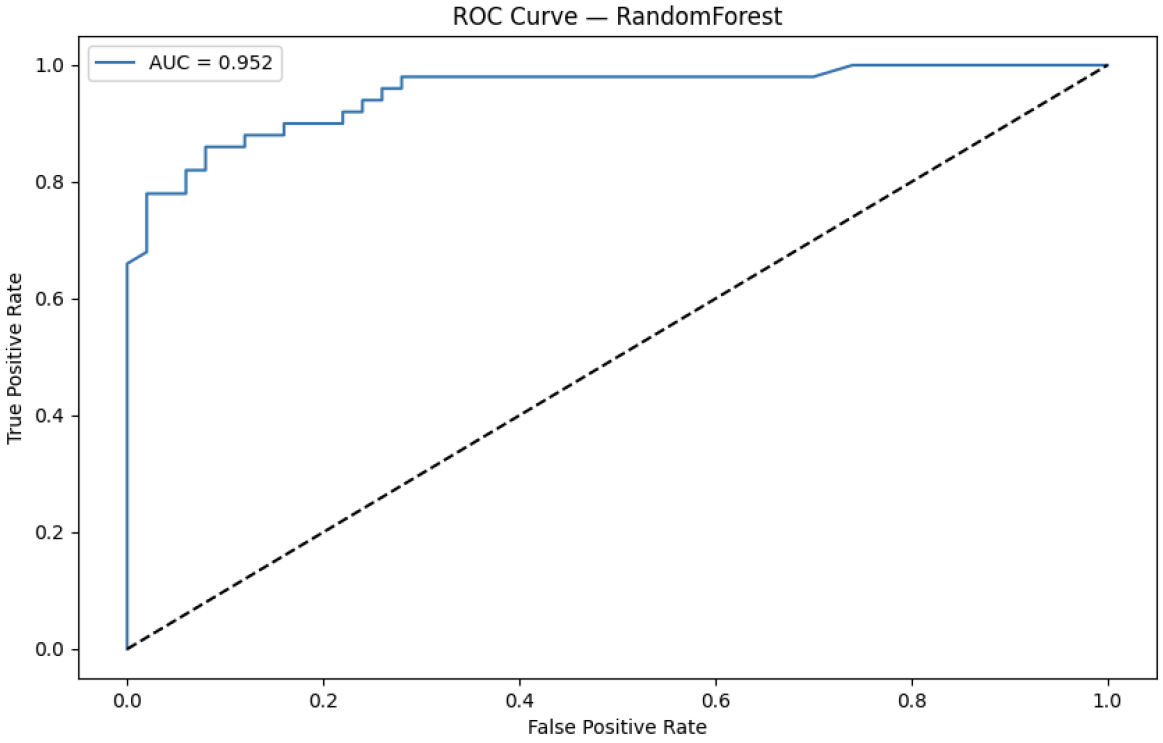
- Değerlerin büyük kısmı 0.83–0.93 aralığında.

- Düşük bir standart sapma, modelin **stabil ve genellenebilir** olduğunu gösterir.

Siber güvenlik yorumu:

- Model, sadece tek bir train–test split'te değil, veri setinin farklı parçalarında da benzer sonuçlar veriyor.
- Bu, "sonuçlar tesadüf değil, veri setinin geneline uygun" demek için önemli bir kanıttır.

4.8 ROC Eğrisi ve AUC Değeri



Şekilde verilen ROC eğrisi, Random Forest modelinin sınıflar arasında yüksek ayrım gücüne sahip olduğunu göstermektedir. Eğrinin sol üst köşeye yakın olması ve AUC = 0.952 değeri, modelin pozitif örnekleri ayırt etmede son derece başarılı olduğunu doğrular.

ROC (Receiver Operating Characteristic) eğrisi, FPR (x ekseni) ve TPR yani recall (y ekseni) arasındaki ilişkiyi, sınıflandırma eşiği (decision threshold) tüm olası değerlerinde değişirken gösterir.

- x ekseni: False Positive Rate (FP / (FP + TN))
- y ekseni: True Positive Rate (TP / (TP + FN)) = Recall

RandomForest için AUC (Area Under Curve):

$$AUC = 0.952$$

AUC'nin anlamı:

- 0.5 → Rasgele tahmin yapan model.
- 1.0 → Mükemmel ayırt edici model.
- 0.95 → Çok yüksek ayrım gücü.

Başka bir deyişle:

> Rasgele seçilen bir malware ile rasgele seçilen bir benign APK'yı karşılaştırdığımızda, modelin malware'e daha yüksek "malware olma skoru" vermesi olasılığı **%95.2**'dir.

Bu, modelin gerçek ve zararlı uygulamaları ayırt etme kapasitesinin çok yüksek olduğunu gösterir.

4.9 XGBoost Sonuçlarıyla Karşılaştırma

XGBoost için confusion matrix:

$$\begin{bmatrix} 45 & 5 \\ 7 & 43 \end{bmatrix}$$

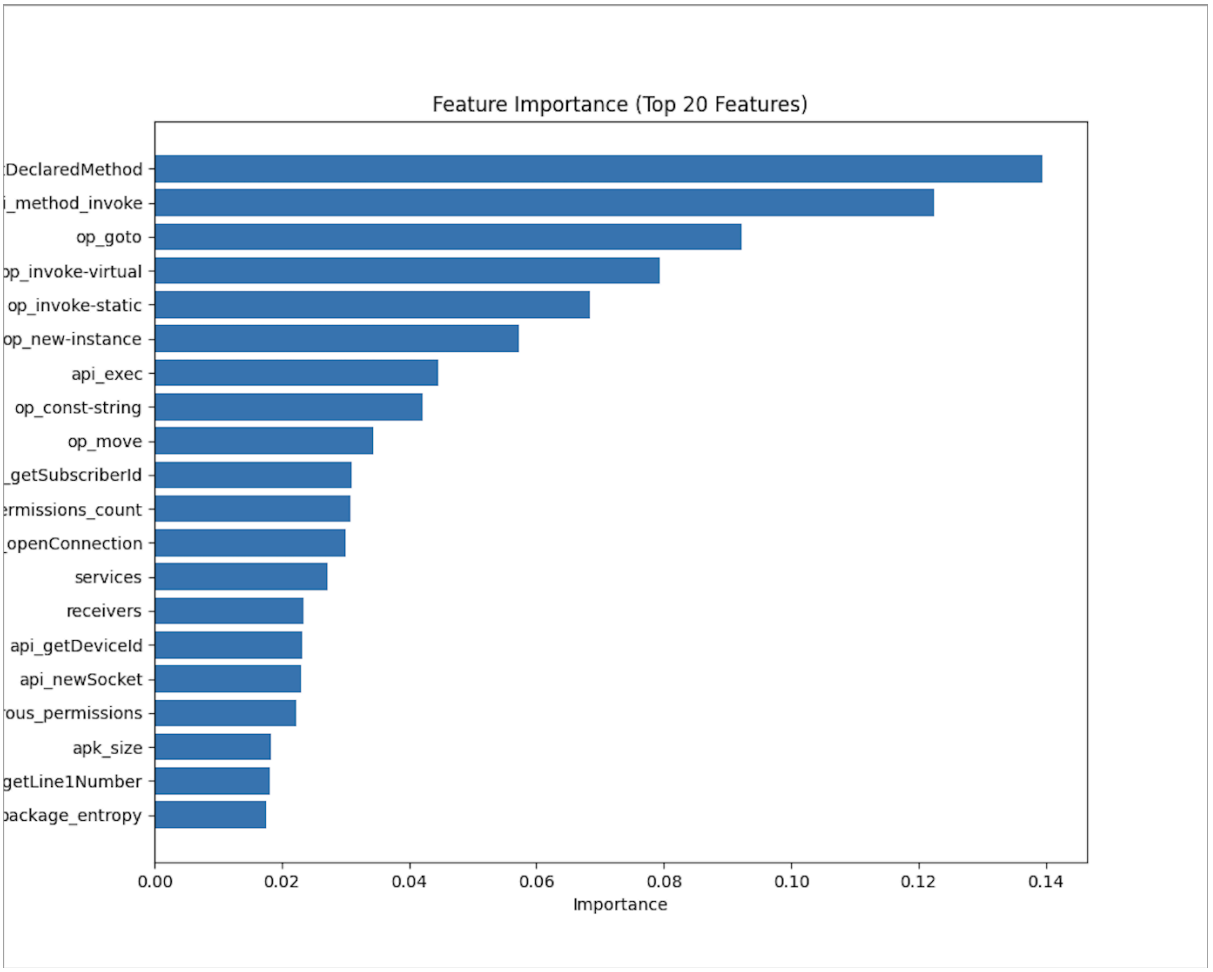
Bu durumda:

- Accuracy yine 0.88
- Malware precision ≈ 0.90
- Malware recall ≈ 0.86

Her iki model de benzer performans sergilemiştir. RandomForest modeli daha basit, yorumlanabilir ve hızlı olduğu için bu proje özelinde ana model olarak tercih edilebilir; XGBoost ise "benchmark / karşılaştırma" modeli olarak rapora ek bilimsel değer katmaktadır.

5. Özellik Önem Düzeyleri ve Korelasyon Analizi (Discussion)

5.1 Feature Importance – Model Neyi Öğrendi?



Şekilde görüldüğü üzere, zararlı yazılımların tespitinde en yüksek katkıyı api_getDeclaredMethod, api_method_invoke, op_goto ve op_invoke-virtual özellikleri sağlamaktadır. Bu durum, modern Android zararlı yazılımlarında reflection ve dinamik kod çağrılarının kritik rol oynadığını göstermektedir.

RandomForest modeli, her özelliğin sınıflandırma kararlarına ne kadar katkı yaptığını **feature_importances_** vektörü ile verir. Grafikte ilk sıralarda görülen özellikler:

1. **api_getDeclaredMethod**
2. **api_method_invoke**
3. **op_goto**
4. **op_invoke-virtual**
5. **op_invoke-static**

6. **op_new-instance**
7. **api_exec**
8. **op_const-string**
9. **api_getSubscriberId**
10. **permissions_count** vb.

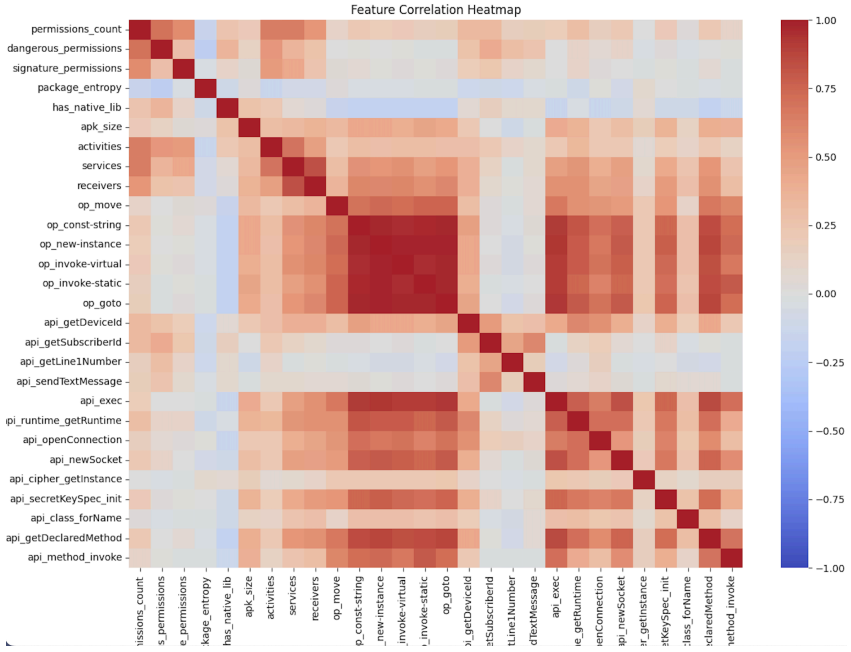
Bu dağılım, modelin **reflection + dinamik çağrı + yoğun çağrı frekansı** üzerinden malware'i tanıdığını göstermektedir.

Reflection API'leri (Class.forName, getDeclaredMethod, invoke):

- Kodun çalışma zamanında string ifadelerle çözümlenmesine imkân tanır.
- Statik analiz araçlarına karşı sıkça kullanılır; çünkü gerçek çağrılar kaynak koda bakıldığında açıkça görünmez.
- Bu yüzden modern zararlı yazılımlarda reflection kullanımı oldukça yaygındır.

Bu çalışmanın en önemli bulgularından biri, **reflection kullanımı arttıkça malware olma olasılığının belirgin şekilde artmasıdır**. RandomForest modeli, bunu otomatik olarak keşfetmiş ve ilgili özelliklere en yüksek önem skorunu vermiştir.

5.2 Korelasyon Heatmap – Özellikler Arasındaki İlişkiler



Şekilde tüm özellikler arasındaki ikili korelasyonları göstermektedir. Özellikle belirli izin grupları ile ağ tabanlı API çağrılarının birlikte yükseldiği gözlemlenmiştir. Ayrıca opcode yoğunluklarının kendi aralarında yüksek korelasyon göstermesi, benzer kontrol akışı yapılarını yansıtmaktadır.

Korelasyon matrisi, her bir özellik çifti arasındaki **Pearson korelasyon katsayısını** gösterir:

- +1 → Tam pozitif ilişki (biri artarken diğeri de artar).
- 0 → İlişki yok.
- -1 → Tam negatif ilişki (biri artarken diğeri azalır).

Isı haritası (heatmap) incelendiğinde:

- op_invoke-virtual, op_invoke-static, op_new-instance gibi çağrı ve nesne oluşturma opcode'larının kendi aralarında yüksek pozitif korelasyona sahip olduğu görülür.
- api_getDeclaredMethod ve api_method_invoke gibi reflection API'leri ile bazı opcode'lar arasında da belirgin ilişkiler vardır.
- İzin sayısı (permissions_count) ile servis / activity sayıları arasında orta seviye korelasyon gözlenir; daha karmaşık uygulamalar daha fazla izin isteme eğilimindedir.

Bu analiz, verinin yapısını anlamak ve gereksiz / tekrar eden (yüksek korelasyonlu) özellikleri tespit etmek açısından önemlidir. Bu projede, feature selection yapılmadı; ancak ileride benzer özellikler gruplanarak model daha da sadeleştirilebilir.

6. Sonuç ve Gelecek Çalışmalar (Conclusion)

Bu çalışma, Android zararlı yazılımlarının yalnızca **statik analiz** ve **makine öğrenmesi** yöntemleriyle etkili biçimde tespit edilebileceğini göstermiştir. AndroZoo'dan alınan 500 APK üzerinde gerçekleştirilen deneylerde:

- RandomForest ve XGBoost modelleri **%88 doğruluk** seviyesine ulaşmıştır.
- RandomForest için **AUC = 0.952** değeri, modelin zararlı ve zararsız uygulamalar arasında yüksek ayırt ediciliğe sahip olduğunu kanıtlamaktadır.
- 10 katlı çapraz doğrulama, modelin farklı veri alt kümelerinde de benzer performans gösterdiğini ve sonuçların tesadüfi olmadığını ortaya koymuştur.
- Özellik önem düzeyi analizi, özellikle reflection temelli API çağrılarının (getDeclaredMethod, invoke) ve yoğun çağrı frekanslarını ifade eden opcode'ların (invoke-*,

goto) zararlı yazılım tespitinde kritik rol oynadığını göstermiştir.

Bu bulgular, statik analiz ile elde edilen zengin özellik setlerinin, doğru seçilmiş makine öğrenmesi modelleri ile birleştirildiğinde yüksek başarımlı bir malware tespit sistemi kurulabileceğini ortaya koymaktadır.

Gelecek çalışmalar için olası geliştirmeler:

1. Veri setinin binlerce APK ile genişletilmesi ve farklı malware ailelerinin ayrı ayrı incelenmesi.

2. Derin öğrenme (deep learning) yaklaşımları ile opcode dizilerinin veya API çağrı dizilerinin doğrudan işlenmesi.

3. Statik ve dinamik analiz bilgilerini birleştiren hibrit bir modeli test etmek.

4. Zamana bağlı analiz: Eski ve yeni malware ailesi davranışlarının karşılaştırılması.