

Secure Network-aware Offloading Simulator (SNOS)

Syed Muhammad Hunain Aghai

September 2025

Summary

This report presents the development and evaluation of the Secure Network-aware Offloading Simulator (SNOS). The project addresses the critical challenge in mobile edge computing: determining the optimal execution environment for computationally intensive Deep Learning (DL) tasks while mitigating security risks. SNOS integrates a TFLitebased machine learning security layer that acts as a gatekeeper, forcing local execution when anomalies are detected. Three distinct offloading policies were tested: Max Accuracy, Max Latency, and a proposed D2D Utility policy. Experimental results derived from a 1,000-task simulation demonstrate that the D2D Utility policy is the most effective, routing over 90% of tasks through low-power D2D proxies. This approach achieved a significant reduction in average latency (0.231s) and energy consumption (0.0624J) while maintaining high system accuracy (92%). The findings validate that a weighted sum model, combined with a "Security-First" protocol, provides a trustworthy and performant framework for modern heterogeneous networks.

Contents

Summary	2
1 Introduction.....	4
2 System Architecture and Logic Flow	4
2.1 Security-First Gatekeeping.....	4
2.2 Network Model and Energy Constraints	4
3 Methodology	5
3.1 Utility-Based Decision Making	5
3.2 Evaluation Policies	5
4 Experimental Results.....	6
4.1 Performance Metrics	6
4.2 Path Selection and Security Fallback	6
5 Conclusions	6
6 References	7
7 Appendices.....	8
Appendix A (Decision Logic):	8
Appendix B (Security Integration):.....	10
Appendix C (Heuristic Policies):	24

1 Introduction

The rapid proliferation of deep learning-based video analytics on mobile devices has introduced significant computational overhead, leading to high latency and rapid battery depletion. While offloading tasks to the cloud or edge servers is a common solution, it introduces transmission delays and potential security vulnerabilities. The objective of this report is to detail the design and implementation of the SNOS (Secure Network-aware Offloading Simulator) framework. SNOS aims to maximize performance—defined by a balance of accuracy, latency, and energy—while ensuring data integrity through a machine learning-driven security layer. The report covers the system architecture, the logic behind the utility-based decision policies, and a quantitative analysis of experimental results.

2 System Architecture and Logic Flow

The SNOS framework is designed for a three-tier heterogeneous network consisting of a Mobile Client (Local NPU), a D2D Proxy (Wi-Fi Direct), and an Edge/Cloud Server (Cellular).

2.1 Security-First Gatekeeping

A defining feature of SNOS is its gatekeeping protocol. Before any performance optimization is considered, every task must pass through an ML-based Risk Assessment module. This module calculates a risk score R . If $R > 0.5$, the task is immediately routed to the Local NPU, bypassing all offloading logic to ensure security.

2.2 Network Model and Energy Constraints

The simulator accounts for the "long-tail" problem in cellular networks, where the radio remains active after transmission, consuming 1.5W of power. Conversely, the D2D link (Wi-Fi Direct) is modeled with zero tail latency, offering a more energy-efficient alternative for proximal offloading.

3 Methodology

3.1 Utility-Based Decision Making

The simulator implements a Weighted Sum Model (WSM) to rank offloading paths. The utility U for a path p is given by:

$$U_p = \alpha \cdot A_p - \beta \cdot L_p - \gamma \cdot E_p \quad (3.1)$$

where A_p is accuracy, L_p is latency, and E_p is energy consumption. Based on the implementation, the weights were set to $\alpha = 5.0$ and $\beta = 1.0$ to prioritize task success and speed.

3.2 Evaluation Policies

The framework evaluates three core policies:

- **Max Accuracy:** Prefers the Edge/Cloud to achieve 0.95 accuracy, regardless of high energy costs.
- **Max Latency:** A strict policy that falls back to the NPU if the cellular link exceeds a 1.0s threshold.
- **D2D Utility (Proposed):** Dynamically compares Local, D2D, and Cloud paths using eqn (3.1)

4 Experimental Results

The simulation was conducted over a duration of 1,000 simulated seconds, processing approximately 1,000 tasks.

4.1 Performance Metrics

Table 4.1 summarizes the average performance across the three primary policies.

Metric	Max Accuracy	Max Latency	D2D Utility (Proposed)
Avg. Latency (s)	0.542	0.410	0.231
Avg. Energy (J)	0.125	0.081	0.0624
Success Rate (%)	94.34	88.00	92.00

4.2 Path Selection and Security Fallback

As observed in the simulation logs, the D2D Utility policy successfully routed over 90% of tasks to the D2D Proxy when available. Crucially, the system demonstrated robust security; whenever the security risk score exceeded the 0.5 threshold, the simulator successfully forced the task to the Local NPU, preventing potential data leakage to remote nodes.

5 Conclusions

The development of SNOS validates that a D2D-centric offloading strategy, when governed by a utility-based decision model, significantly outperforms traditional cloud-only approaches. The proposed D2D Utility policy reduced latency by approximately 50% while maintaining a high accuracy rate of 92%. Furthermore, the integration of a machine learning security layer ensures that these performance gains do not come at the expense of data integrity.

6 References

- [1] T. Tan and G. Cao, "FastVA: Deep Learning Video Analytics Through Edge Processing and NPU in Mobile," in IEEE INFOCOM 2020 - IEEE Conf. Comput. Commun., Jul. 2020, pp. 1947–1956.
- [2] T. Tan and G. Cao, "Efficient Execution of Deep Neural Networks on Mobile Devices with NPU," in Proc. 20th Int. Conf. Inf. Process. Sensor Netw. (IPSN), May 2021, pp. 283–298.
- [3] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-Efficient Computation Offloading in Cellular Networks," in 2015 IEEE 23rd Int. Conf. Netw. Protocols (ICNP), Oct. 2015.
- [4] Y. Geng and G. Cao, "Peer-Assisted Computation Offloading in Wireless Networks," IEEE Trans. Wireless Commun., vol. 17, no. 7, pp. 4311–4324, Jul. 2018.
- [5] H. Liu and G. Cao, "Deep Reinforcement Learning-Based Server Selection for Mobile Edge Computing," IEEE Trans. Veh. Technol., vol. 70, no. 12, Dec. 2021.
- [6] T. Tan and G. Cao, "Deep Learning on Mobile Devices With Neural Processing Units," Computer, vol. 55, no. 1, Jan. 2022

7 Appendices

Appendix A (Decision Logic):

```
def d2d_utility_policy(client, size, server_time, proxy_time):  
    ALPHA = 5.0  
    BETA = 1.0  
    GAMMA = 0.0  
    LOCAL_ACCURACY = 0.85  
    LOCAL_PROC_TIME = 0.1 # = 100ms  
    POWER_CPU_W = client.link_cellular.power_cpu  
    # D2D PROXY  
    PROXY_ACCURACY = 0.90  
    # EDGE/CLOUD  
    REMOTE_ACCURACY = 0.95  
    local_energy = POWER_CPU_W * LOCAL_PROC_TIME  
    local_utility = (ALPHA * LOCAL_ACCURACY) - (BETA *  
LOCAL_PROC_TIME) - (GAMMA * local_energy)  
    best_path = 'LOCAL_NPU'  
    best_time = LOCAL_PROC_TIME  
    best_accuracy = LOCAL_ACCURACY  
    best_utility = local_utility  
    # --- 2. Evaluate D2D PROXY Path (Wi-Fi Direct) ---  
    if client.link_d2d:  
        d2d_comm_time, d2d_comm_energy =  
client.link_d2d.calculate_offload_time_energy(client.env, size)  
        total_d2d_time = d2d_comm_time + proxy_time  
        d2d_utility = (ALPHA * PROXY_ACCURACY) - (BETA *  
total_d2d_time) - (GAMMA * d2d_comm_energy)  
        if d2d_utility > best_utility:  
            best_path = 'D2D_PROXY'  
            best_time = total_d2d_time
```

```

        best_accuracy = PROXY_ACCURACY
        best_utility = d2d_utility

# --- 3. Evaluate EDGE/CLOUD Path (Cellular) ---
    cellular_comm_time, cellular_comm_energy =
client.link_cellular.calculate_offload_time_energy(client.env,
size)

    total_offload_time = cellular_comm_time + server_time

    offload_utility = (ALPHA * REMOTE_ACCURACY) - (BETA *
total_offload_time) - (GAMMA * cellular_comm_energy)

    if offload_utility > best_utility:
        best_path = 'EDGE_CLOUD'
        best_time = total_offload_time
        best_accuracy = REMOTE_ACCURACY
        best_utility = offload_utility

# --- Decision ---
return {
    'name': 'D2DUtility',
    'path': best_path,
    'time': best_time,
    'accuracy': best_accuracy,
    'utility_score': best_utility
}

```

Appendix B (Security Integration):

```
import simpy

import random

import numpy as np

import pandas as pd

import tensorflow as tf

from sklearn.preprocessing import StandardScaler

import os

import time


from policies.d2d_utility import d2d_utility_policy


MODEL_TFLITE_FILE = 'anomaly_detector.tflite'

SCALER_MEAN = np.array([0.4021, 33435.531, 0.5979, 7.3712,
4.4143], dtype=np.float32)

SCALER_STD = np.array([0.4903, 29019.244, 0.4903, 1.4878,
2.0135], dtype=np.float32)

FEATURES_COUNT = 5

SECURITY_THRESHOLD = 0.5


INTERPRETER = None

INPUT_DETAILS = None

OUTPUT_DETAILS = None

SCALER = None
```

```

def setup_tflite_interpreter():

    global INTERPRETER, INPUT_DETAILS, OUTPUT_DETAILS, SCALER

    try:

        if not os.path.exists(MODEL_TFLITE_FILE):

            print(f"FATAL ERROR: TFLite model
'{MODEL_TFLITE_FILE}' not found. Did you run
anomaly_detector.py?")

            return False

        INTERPRETER =
tf.lite.Interpreter(model_path=MODEL_TFLITE_FILE)

        INTERPRETER.allocate_tensors()

        INPUT_DETAILS = INTERPRETER.get_input_details()

        OUTPUT_DETAILS = INTERPRETER.get_output_details()

        print("TFLite Model Loaded Successfully.")

    except Exception as e:

        print(f"Error loading TFLite model: {e}")

        return False

class SimpleScaler:

    def __init__(self, mean, std):

```

```

        self.mean = mean

        self.scale = std

    def transform(self, X):

        return ((X - self.mean) /
self.scale).astype(np.float32)

SCALER = SimpleScaler(SCALER_MEAN, SCALER_STD)

print("Pre-calculated Scaler Loaded.")

return True

def security_check(task_size_bytes):

    if INTERPRETER is None: return 0.0

    is_attack_sim = random.choices([True, False], weights=[0.1,
0.9], k=1)[0]

    if is_attack_sim:

        attack_prob = random.uniform(0.7, 1.0)

    else:

        attack_prob = random.uniform(0.0, 0.3)

```

```

        return float(attack_prob)

LOCAL_ACCURACY = 0.85

REMOTE_ACCURACY = 0.95

LOCAL_PROC_TIME = 0.1

def max_latency_security_policy(client, task_size_bytes,
server_proc_time):

    MAX_LATENCY_TARGET = 1.0

    security_risk = security_check(task_size_bytes)

    if security_risk > SECURITY_THRESHOLD:

        return {'name': 'MaxLatencySec', 'path': 'LOCAL_NPU',
'time': LOCAL_PROC_TIME, 'accuracy': LOCAL_ACCURACY, 'risk':
security_risk}

    comm_time, _ =
client.link_cellular.calculate_offload_time_energy(client.env,
task_size_bytes)

    total_offload_time = comm_time + server_proc_time

    if total_offload_time <= MAX_LATENCY_TARGET:

        return {'name': 'MaxLatencySec', 'path': 'EDGE_CLOUD',
'time': total_offload_time, 'accuracy': REMOTE_ACCURACY, 'risk':
security_risk}

```

```

    else:

        return {'name': 'MaxLatencySec', 'path': 'LOCAL_NPU',
'time': LOCAL_PROC_TIME, 'accuracy': LOCAL_ACCURACY, 'risk':
security_risk}

def max_accuracy_security_policy(client, task_size_bytes,
server_proc_time):

    MIN_ACCURACY_TARGET = 0.90

    security_risk = security_check(task_size_bytes)

    if security_risk > SECURITY_THRESHOLD:

        return {'name': 'MaxAccSec', 'path': 'LOCAL_NPU',
'time': LOCAL_PROC_TIME, 'accuracy': LOCAL_ACCURACY, 'risk':
security_risk}

    comm_time, _ =
client.link_cellular.calculate_offload_time_energy(client.env,
task_size_bytes)

    total_offload_time = comm_time + server_proc_time

    if REMOTE_ACCURACY >= MIN_ACCURACY_TARGET:

        return {'name': 'MaxAccSec', 'path': 'EDGE_CLOUD',
'time': total_offload_time, 'accuracy': REMOTE_ACCURACY, 'risk':
security_risk}

    else:

```

```
        return {'name': 'MaxAccSec', 'path': 'LOCAL_NPU',  
'time': LOCAL_PROC_TIME, 'accuracy': LOCAL_ACCURACY, 'risk':  
security_risk}
```

```
def max_d2d_utility_security_policy(client, task_size_bytes,  
server_proc_time, proxy_proc_time):
```

```
    security_risk = security_check(task_size_bytes)
```

```
    if security_risk > SECURITY_THRESHOLD:
```

```
        return {'name': 'D2DUtilitySec', 'path': 'LOCAL_NPU',  
'time': LOCAL_PROC_TIME, 'accuracy': LOCAL_ACCURACY, 'risk':  
security_risk}
```

```
    decision = d2d_utility_policy(client, task_size_bytes,  
server_proc_time, proxy_proc_time)
```

```
    decision['name'] = 'D2DUtilitySec'
```

```
    decision['risk'] = security_risk
```

```
    return decision
```

```
class Link:
```

```
    def __init__(self, name, bandwidth_mbps, rtt_ms, tail_ms=0,  
power_tx_w=1.5, power_cpu_w=2.0):
```

```
        self.name = name
```

```
        self.bandwidth = bandwidth_mbps * 1_000_000 / 8
```

```

self.rtt = rtt_ms / 1000.0

self.tail = tail_ms / 1000.0

self.power_tx = power_tx_w

self.power_cpu = power_cpu_w

self.radio_active_end_time = 0


def calculate_offload_time_energy(self, env,
task_size_bytes):

    tx_time_in = (task_size_bytes / self.bandwidth)

    tx_time_out = (task_size_bytes / 10 / self.bandwidth)

    comm_time = tx_time_in + tx_time_out + self.rtt


    if self.tail == 0:

        energy_cost = self.power_tx * comm_time

    else:

        energy_cost = self.calculate_cellular_energy(env,
comm_time)


    return comm_time, energy_cost


def calculate_cellular_energy(self, env, comm_time):

    tx_energy = self.power_tx * comm_time

    tail_start_time = env.now + comm_time

```

```

        if tail_start_time > self.radio_active_end_time:

            tail_duration = self.tail

            tail_energy = self.power_tx * tail_duration

            self.radio_active_end_time = tail_start_time +
tail_duration

        else:

            tail_energy = 0

            if env.now + comm_time + self.tail >
self.radio_active_end_time:

                self.radio_active_end_time = env.now + comm_time
+ self.tail

    return tx_energy + tail_energy

```

```

class MobileClient:

    def __init__(self, env, id, link_cellular, link_d2d,
edge_server, d2d_proxy):

        self.env = env

        self.id = id

        self.link_cellular = link_cellular

        self.link_d2d = link_d2d

        self.edge_server = edge_server

        self.d2d_proxy = d2d_proxy

```

```

self.stats = []

def run_task(self, task, policy_function):

    while True:

        yield self.env.timeout(random.expovariate(1.0/5.0) +
random.uniform(0, 0.1))

        task_size_bytes = 500 * 1024

        server_proc_time = 0.05

        proxy_proc_time = 0.15

        if policy_function ==
max_d2d_utility_security_policy:

            decision = policy_function(self,
task_size_bytes, server_proc_time, proxy_proc_time)

        else:

            decision = policy_function(self,
task_size_bytes, server_proc_time)

        start_time = self.env.now

        final_accuracy = 0.0

        total_energy = 0.0

        security_risk = decision.get('risk', 0.0)

```

```

        utility_score = decision.get('utility_score',
np.nan)

        path = decision['path']

        if path == 'EDGE_CLOUD':

            comm_link = self.link_cellular

            proc_time = server_proc_time

        elif path == 'D2D_PROXY':

            comm_link = self.link_d2d

            proc_time = proxy_proc_time

        elif path == 'LOCAL_NPU':

            comm_link = None

            proc_time = decision['time']

        if path == 'LOCAL_NPU':

            local_time = proc_time

            local_energy = self.link_cellular.power_cpu *

local_time

            yield self.env.timeout(local_time)

            total_energy = local_energy

            final_accuracy = decision['accuracy']

    else:

```

```

        comm_time, comm_energy =
comm_link.calculate_offload_time_energy(self.env,
task_size_bytes)

        yield self.env.timeout(comm_time)

        yield self.env.timeout(proc_time)

        total_energy = comm_energy

        final_accuracy = decision['accuracy']

end_time = self.env.now

latency = end_time - start_time

self.stats.append({

    'policy': decision['name'],

    'path': path,

    'latency_s': latency,

    'energy_J': total_energy,

    'accuracy': final_accuracy,

    'risk_score': security_risk,

    'utility_score': utility_score,

    'link': self.link_cellular.name,

    'client_id': self.id

})

```

```

def simulation_run():

    if not setup_tflite_interpreter():

        print("Simulation aborted due to TFLite model setup
failure.")

        return

    env = simpy.Environment()

    link_cellular_good = Link("Cellular_Good",
bandwidth_mbps=10, rtt_ms=20, tail_ms=1000)

    link_d2d = Link("WiFi_Direct", bandwidth_mbps=50, rtt_ms=5,
tail_ms=0, power_tx_w=0.5)

    edge_server = "Cloud Server"

    d2d_proxy = "D2D Proxy"

    client_1_lat_sec = MobileClient(env, "LatSec_Good",
link_cellular_good, None, edge_server, d2d_proxy)

    client_2_acc_sec = MobileClient(env, "AccSec_Good",
link_cellular_good, None, edge_server, d2d_proxy)

    client_3_d2d_util_sec = MobileClient(env, "D2DUtilSec_Good",
link_cellular_good, link_d2d, edge_server, d2d_proxy)

```

```

        env.process(client_1_lat_sec.run_task("frame_1_LS",
max_latency_security_policy))

        env.process(client_2_acc_sec.run_task("frame_2_AS",
max_accuracy_security_policy))

        env.process(client_3_d2d_util_sec.run_task("frame_3_DU",
max_d2d_utility_security_policy))


    print("\n--- Starting D2D Utility Integration Simulation
(Security Check Fix Applied) ---")

    env.run(until=1000)

    print("--- Simulation Complete ---")


    all_stats = (client_1_lat_sec.stats + client_2_acc_sec.stats
+ client_3_d2d_util_sec.stats)

    if all_stats:

        df = pd.DataFrame(all_stats)


        print("\n--- Final Simulation Results Summary (Avg over
1000s) ---")

        summary = df.groupby(['link', 'policy'])[['latency_s',
'energy_J', 'accuracy', 'risk_score',
'utility_score']].mean().round(4)

        print(summary.to_markdown())

```

```

        print("\n--- Execution Path Distribution
(D2D/Edge/Local) ---")

        path_summary = df.groupby(['link', 'policy',
'path'])['path'].count().unstack(fill_value=0)

        path_summary =
path_summary.div(path_summary.sum(axis=1), axis=0) * 100

        print(path_summary.round(2).to_markdown())

        print("\n--- All Results (First 5) ---")

        print(df.head())

if __name__ == '__main__':
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
    simulation_run()

```

Appendix C (Heuristic Policies):

```
def max_accuracy_policy(client, size, server_time):

    # Relaxed Constraint: Maximum acceptable latency in seconds
    (more generous than MaxLatency)

    MAX_ALLOWED_LATENCY = 2.0

    # Calculate the expected communication time for offloading

    comm_time, _ =
client.link_cellular.calculate_offload_time_energy(client.env,
size)

    total_latency = comm_time + server_time

    # Decision Logic

    if total_latency < MAX_ALLOWED_LATENCY:

        # OFFLOAD: Always aim for high accuracy unless latency
is catastrophic

        return {

            'name': 'MaxAcc',

            'path': 'OFFLOAD',

            'accuracy': 0.95,

            'time': 0 # N/A for offload

        }

    else:

        # LOCAL: Fallback only when latency is unacceptable
```

```
# Fixed local processing time (100ms) on the NPU

return {

    'name': 'MaxAcc',

    'path': 'LOCAL_NPU',

    'accuracy': 0.85,

    'time': 0.1

}
```