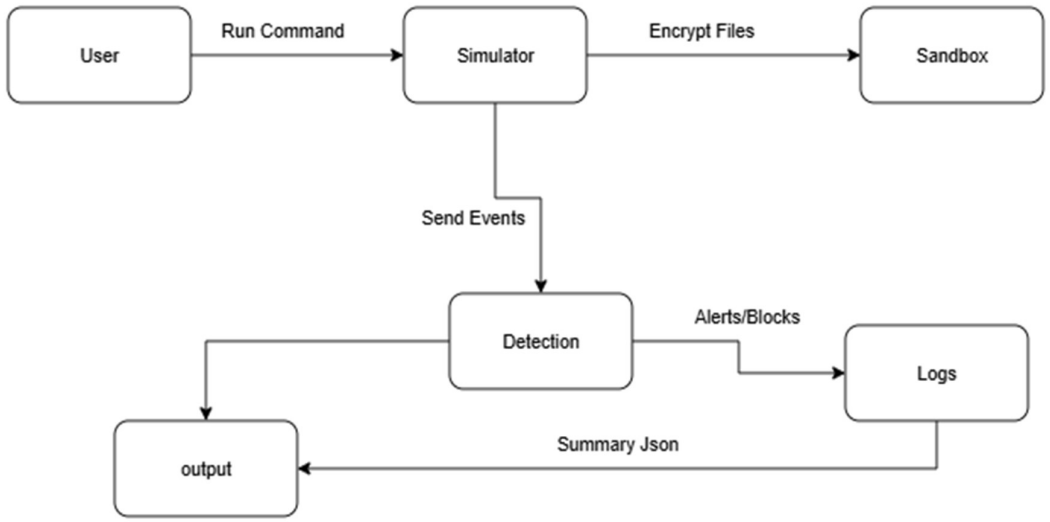
DEPARTMENT	**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**				
Semester: VII	**Course Title: Data Security & Privacy Lab**				**Course Code: 22AI73**
PROGRAM QUESTION	3. Build a virus simulation tool — safe, educational, and non-malicious — to help you understand how malware might behave in a system, and how defenders can detect or prevent it.				
STUDENT NAMES	AYUSH MEHTA		USN	1DS22AI008	
LAB MARKS	**Program Execution (10)**	**Source Code (10)**	**Result (5)**	**Innovation/ Extra Features (5)**	**Total**
TOOLS USED					
MAP COURSE OUTCOME	**CO1**: Understand the basic behavior of different types of malware (ransomware, worm, keylogger, etc.) in a controlled environment. **CO2**: Analyze system activities (file creation, renaming, logging) and relate them to malicious behaviors. **CO3**: Demonstrate how detection and prevention mechanisms (alerts, blocks, quarantines) work. **CO4**: Apply safe simulation tools to study malware without real-world risks.				
MAP PROGRAM OUTCOME	**PO1 (Engineering Knowledge):** Apply knowledge of computer science and cybersecurity to simulate malware behavior. **PO2 (Problem Analysis):** Identify malicious patterns like rapid file encryption or mass note creation. **PO3 (Design/Development of Solutions):** Design a safe sandbox simulation environment to study malware. **PO4 (Conduct Investigations):** Interpret logs and alerts to evaluate malware detection strategies.				
MAP PROGRAM SPECIFIC OUTCOME	**PSO1:** Ability to apply cybersecurity principles to model and analyze malware safely. **PSO2:** Ability to design controlled experiments for testing security tools and defensive strategies. **PSO3:** Ability to interpret detection and prevention data to recommend improvements in system defense.				

UML / SEQUENCE DIGRAM	
COURSE COORDINATOR	Dr ARUNA M G & Prof ENSTEIH SILVIA

Code Implementation

Simulator.py

```
#!/usr/bin/env python3
"""
SAFE Malware Behavior Simulator (educational)
- Runs entirely in a sandbox directory you choose
- Simulates common malware *behaviors* without using real exploits or persistence
- Emits rich telemetry so you can practice building detections & mitigations
- Includes optional "prevent" controls to block a scenario when a rule triggers

⚠️ Safety rails:
- Refuses to run outside a dedicated sandbox directory
- Refuses to run as root/admin
- Never touches files it didn't create
- No real networking, registry, drivers, autoruns, or privilege changes
- Scenarios use fake data only

Usage:
  python simulator.py --sandbox ./sandbox --scenario ransomware --duration 8s
  python simulator.py --sandbox ./sandbox --scenario keylogger --duration 8s
  python simulator.py --sandbox ./sandbox --scenario worm --duration 8s
  python simulator.py --sandbox ./sandbox --scenario benign --duration 8s
  python simulator.py --sandbox ./sandbox --scenario ransomware --duration 8s --prevent

You must also pass the acknowledgment flag:
  --i-understand-this-is-a-simulation
"""

import argparse
import os
import sys
import time
import json
import base64
import random
import string
from pathlib import Path
from dataclasses import dataclass, field
from typing import List, Dict, Any
import threading

# ---------- Safety checks ----------
```

```
def is_admin() -> bool:
    try:
        return os.geteuid() == 0  # POSIX
    except AttributeError:
        # On Windows, avoid admin checks; this tool is purely safe either way
        return False

def ensure_safe_sandbox(path: Path):
    if not path.exists():
        path.mkdir(parents=True, exist_ok=True)
    # Only allow if path is empty OR contains a marker file from prior runs
    marker = path / ".SAFE_SIM_SANDBOX"
    if any(path.iterdir()) and not marker.exists():
        raise SystemExit(f"Refusing to run: sandbox '{path}' is not empty and lacks marker. Choose
an empty dir.")
    marker.touch()
    # Build safe subdirs
    for sub in ["data", "logs", "hosts", "quarantine"]:
        (path / sub).mkdir(exist_ok=True)

# ---------- Eventing / Telemetry ----------

@dataclass
class Event:
    ts: float
    kind: str
    details: Dict[str, Any] = field(default_factory=dict)

class EventBus:
    def __init__(self, out: Path):
        self.events: List[Event] = []
        self.out = out

    def emit(self, kind: str, **details):
        evt = Event(ts=time.time(), kind=kind, details=details)
        self.events.append(evt)

    def flush(self):
        # Write JSONL for easy ingestion
        with open(self.out, "w", encoding="utf-8") as f:
            for e in self.events:
                f.write(json.dumps({"ts": e.ts, "kind": e.kind, **e.details}) + "\n")

# ---------- Scenarios (all SAFE & local only) ----------
```

```
class Scenario:
    def __init__(self, sandbox: Path, bus: EventBus, stop_flag: threading.Event):
        self.sandbox = sandbox
        self.bus = bus
        self.stop_flag = stop_flag

    def wait(self, secs: float):
        # Small sleeps and cooperative stop
        end = time.time() + secs
        while time.time() < end:
            if self.stop_flag.is_set():
                return
            time.sleep(0.05)

    def run(self, duration_s: float):
        raise NotImplementedError

class BenignScenario(Scenario):
    def run(self, duration_s: float):
        data = self.sandbox / "data"
        # Simulate normal app making a few files and logs
        for i in range(5):
            if self.stop_flag.is_set(): break
            p = data / f"report_{i}.txt"
            txt = f"Quarterly report line {i} at {time.time()}\\n"
            p.write_text(txt, encoding="utf-8")
            self.bus.emit("file_write", path=str(p), bytes=len(txt))
            self.wait(0.2)
        # Occasional config read
        for i in range(3):
            if self.stop_flag.is_set(): break
            self.bus.emit("config_read", path=str(self.sandbox / 'config.ini'))
            self.wait(duration_s / 10)

class RansomwareScenario(Scenario):
    def run(self, duration_s: float):
        data = self.sandbox / "data"
        # Create dummy files
        for i in range(30):
            if self.stop_flag.is_set(): return
            p = data / f"doc_{i}.txt"
            content = f"SAFE-DUMMY-FILE-{i}-" + ("".join(random.choices(string.ascii_letters,
k=200)))
            p.write_text(content, encoding="utf-8")
        self.bus.emit("seed_files_created", count=30)
```

```
        # "Encrypt" (harmlessly transform) the files we just made
        start = time.time()
        while time.time() - start < duration_s and not self.stop_flag.is_set():
            for p in list(data.glob("*.txt")):
                if self.stop_flag.is_set(): break
                payload = p.read_text(encoding="utf-8")
                # Reversible transformation: base64 (safe, not destructive)
                transformed = base64.b64encode(payload.encode("utf-8")).decode("ascii")
                locked = p.with_suffix(p.suffix + ".simlocked")
                locked.write_text(transformed, encoding="utf-8")
                p.unlink()  # remove original (still only our files)
                self.bus.emit("file_encrypt_sim", src=str(p), dst=str(locked), method="base64")
                self.wait(0.01)
            # Write a fake ransom note *inside* sandbox
            note = self.sandbox / "NOTE_README_SIM.txt"
            note.write_text("This is a SAFE simulation. No real encryption occurred.", encoding="utf-
8")
            self.bus.emit("ransom_note_write", path=str(note))
            self.wait(0.2)

class KeyloggerScenario(Scenario):
    def run(self, duration_s: float):
        # Generates fake keystrokes from a predefined string; no hooking
        fake_text = "This is a SAFE simulated keystroke stream for blue-team training. "
        log = self.sandbox / "logs" / "keystrokes.log"
        start = time.time()
        i = 0
        while time.time() - start < duration_s and not self.stop_flag.is_set():
            chunk = fake_text[i % len(fake_text)]
            with open(log, "a", encoding="utf-8") as f:
                f.write(chunk)
            self.bus.emit("keystroke_write_sim", char=chunk, path=str(log))
            i += 1
            self.wait(0.02)

class WormScenario(Scenario):
    def run(self, duration_s: float):
        # Simulate scanning and "infecting" localhost-like fake hosts by writing markers
        hosts_dir = self.sandbox / "hosts"
        hosts = [hosts_dir / f"host_{i}" for i in range(1, 16)]
        for h in hosts:
            h.mkdir(exist_ok=True)
        start = time.time()
        idx = 0
        while time.time() - start < duration_s and not self.stop_flag.is_set():
            target = hosts[idx % len(hosts)]
```

```
            marker = target / "infected.txt"
            if not marker.exists():
                marker.write_text("SAFE_SIM_INFECTED_MARKER", encoding="utf-8")
                self.bus.emit("propagate_sim", host=target.name, path=str(marker))
            # Beacon (simulated) to controller (no real network)
            self.bus.emit("beacon_sim", endpoint="cnc.local", status="OK")
            idx += 1
            self.wait(0.1)

# ---------- Detection / Prevention ----------

@dataclass
class Rule:
    name: str
    when: str
    threshold: int
    window_seconds: float
    action: str  # "alert" or "block"

def load_rules(path: Path) -> List[Rule]:
    # Minimal YAML-ish loader to avoid extra deps
    import re
    text = path.read_text(encoding="utf-8")
    blocks = [b.strip() for b in re.split(r'^-\\s*$', text, flags=re.M)]
    rules = []
    cur = {}
    for line in text.splitlines():
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        if line.startswith("-"):
            if cur:
                rules.append(Rule(**cur))
                cur = {}
            continue
        if ":" in line:
            k, v = line.split(":", 1)
            k = k.strip()
            v = v.strip()
            if k in {"threshold"}:
                cur[k] = int(v)
            elif k in {"window_seconds"}:
                cur[k] = float(v)
            else:
                cur[k] = v
    if cur:
```

```
        rules.append(Rule(**cur))
    return rules

def evaluate_rules(events: List[Event], rules: List[Rule]):
    alerts = []
    blocks = []
    for rule in rules:
        # sliding window count for matching event kind
        times = [e.ts for e in events if e.kind == rule.when]
        times.sort()
        i = 0
        for j in range(len(times)):
            while times[j] - times[i] > rule.window_seconds:
                i += 1
            count = j - i + 1
            if count >= rule.threshold:
                alerts.append(rule.name)
                if rule.action == "block":
                    blocks.append(rule.name)
                break
    return alerts, blocks

# ---------- Orchestrator ----------

def main():
    ap = argparse.ArgumentParser(description="SAFE Malware Behavior Simulator
(educational)")
    ap.add_argument("--sandbox", required=True, help="Dedicated empty directory for the
simulation")
    ap.add_argument("--scenario", required=True, choices=["benign", "ransomware", "keylogger",
"worm"])
    ap.add_argument("--duration", default="8s", help="Duration like 5s, 2m (caps to scenario
where relevant)")
    ap.add_argument("--rules", default="rules.yml", help="Detection rules file")
    ap.add_argument("--prevent", action="store_true", help="Enable prevention: stop scenario
when a block rule fires")
    ap.add_argument("--i-understand-this-is-a-simulation", action="store_true", dest="ack",
                    help="Required safety acknowledgment")
    args = ap.parse_args()

    if not args.ack:
        raise SystemExit("Refusing to run without --i-understand-this-is-a-simulation")

    if is_admin():
        raise SystemExit("Refusing to run as root/admin. Use a normal user.")
```

```
    # Parse duration
    dur_s = 8.0
    ds = args.duration.strip().lower()
    if ds.endswith("ms"):
        dur_s = max(0.05, float(ds[:-2]) / 1000.0)
    elif ds.endswith("s"):
        dur_s = max(0.1, float(ds[:-1]))
    elif ds.endswith("m"):
        dur_s = max(0.1, float(ds[:-1]) * 60.0)
    else:
        dur_s = max(0.1, float(ds))

    sandbox = Path(args.sandbox).resolve()
    ensure_safe_sandbox(sandbox)

    bus = EventBus(out=sandbox / "logs" / "events.jsonl")

    stop_flag = threading.Event()
    scenarios = {
        "benign": BenignScenario,
        "ransomware": RansomwareScenario,
        "keylogger": KeyloggerScenario,
        "worm": WormScenario,
    }
    scen = scenarios[args.scenario](sandbox, bus, stop_flag)

    # Run scenario (in thread so we can stop it on prevention)
    th = threading.Thread(target=scen.run, args=(dur_s,), daemon=True)
    th.start()
    th.join(timeout=dur_s + 2.0)  # best-effort join

    # Flush events and run detections
    bus.flush()
    rules_path = Path(args.rules)
    if not rules_path.exists():
        print(f"[WARN] Rules file not found at {rules_path}; skipping detection.")
        return 0

    rules = load_rules(rules_path)
    alerts, blocks = evaluate_rules(bus.events, rules)

    summary = {
        "scenario": args.scenario,
        "sandbox": str(sandbox),
        "event_count": len(bus.events),
        "alerts_triggered": alerts,
```

```
        "blocks_triggered": blocks,
        "prevention_enabled": args.prevent,
    }

    # Prevention: if any block rules matched, signal stop (for next run) and quarantine any new files
we created
    if args.prevent and blocks:
        stop_flag.set()
        # Quarantine any *.simlocked files and keystrokes log
        quarantine = sandbox / "quarantine"
        data = sandbox / "data"
        for p in list(data.glob("*.simlocked")) + [sandbox / "logs" / "keystrokes.log"]:
            if p.exists():
                qp = quarantine / p.name
                try:
                    p.replace(qp)
                    bus.emit("quarantine", path=str(qp))
                except Exception:
                    pass
        bus.flush()
        summary["prevention_action"] = "quarantined_files"

    # Write summary
    (sandbox / "logs" / "summary.json").write_text(json.dumps(summary, indent=2),
encoding="utf-8")
    print(json.dumps(summary, indent=2))
    return 0

if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("Interrupted.")
        sys.exit(1)
```

Output

```
PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool> python -m venv venv
PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool> venv\Scripts\activate
(venv) PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool> python simulator.py
usage: simulator.py [-h] --sandbox SANDBOX --scenario {benign,ransomware,keylogger,worm} [--duration DURATION] [--rules RULES] [--prevent] [--i-understand-this-is-a-simulation]
simulator.py: error: the following arguments are required: --sandbox, --scenario
(venv) PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool> python simulator.py --sandbox ./sandbox --scenario ransomware --duration 8s --i-understand-this-is-a-simulation
>>
{
  "scenario": "ransomware",
  "sandbox": "C:\\Users\\ASUS\\Downloads\\safe_malware_simulator\\malware_sim_tool\\sandbox",
  "event_count": 60,
  "alerts_triggered": [
    "\"Potential ransomware: rapid 'encryption' of many files\"",
    "\"Block on ransom-note creation flood\""
  ],
  "blocks_triggered": [
    "\"Block on ransom-note creation flood\""
  ],
  "prevention_enabled": false
}
(venv) PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool>
(venv) PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool>
(venv) PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool>
 * History restored

PS C:\Users\ASUS\Downloads\safe_malware_simulator\malware_sim_tool>
```