Application of the Python programming language in computer forensics

Valentina Petrova
Nikola Vaptsarov Naval Academy, Varna, Bulgaria
v.petrova@naval-acad.bg

Abstract: *This paper presents the application of Python in computer forensics, proposing various approaches for file system analysis, memory forensics, network monitoring, and malware detection. It examines Python's integration with forensic tools such as Autopsy, The Sleuth Kit (TSK), Volatility, and Scapy, highlighting its role in automating forensic investigations and enhancing efficiency. The research discusses the advantages of Python in forensic analysis, emphasizing its role in forensic automation, script-based evidence collection, and rapid data processing within various case studies.*
Keywords*: COMPUTER FORENSIC TOOLS, COMPUTER FORENSICS*

1. Introduction

Computer forensics is the branch of forensic science dedicated to identifying, preserving, analyzing, and presenting digital evidence from various electronic devices. The primary objective is to recover data, analyze system artifacts, and provide evidence for legal proceedings or incident response [1,2].

Python has become a fundamental programming language in the digital forensics' community due to its simplicity, flexibility, and wide range of libraries. It facilitates the automation of repetitive tasks, the development of custom modules, and the integration of forensic tools into cohesive workflows. Python's ability to interact with various forensic tools makes it the instrument for improving the efficiency and effectiveness of forensic investigations.

Python allows users to saving time and reducing the human errors. The investigators can enhance their analysis with specialized plugins or scripts. The programming language can easily interface with other tools, databases, and services, making it an excellent choice for integrating different forensic tools into a single workflow.

2. Computer Forensic Tools and case studies

Python scripts can automate the process of running commands and processing the results in computer forensic tools. Python automates the extraction of file metadata, perform searches for specific keywords, and generate reports without needing to manually interact with command-line interface.

TSK has a Python bindings library (PyTSK) that allows users to interact with disk images and filesystem structures programmatically. Python scripts can be written to parse filesystem data, recover deleted files, analyze file slack space, and explore partition structures. Python can be used to integrate TSK with other forensic tools or external data sources, creating a more comprehensive and customized workflow. The programming code imports data from forensic tools into a database, or to integrate with other tools like Volatility (memory forensics) or external analysis tools (e.g., hashing algorithms, antivirus engines).

Autopsy is a powerful open-source digital forensic tool built on The Sleuth Kit (TSK). While it provides a graphical interface, Autopsy also has a Python API that allows users to automate investigations, extract artifacts, and extend functionality. Below are practical case studies demonstrating how to integrate Python with Autopsy for digital forensics. Autopsy provides an extensible framework that allows developers to create custom modules and tools using Python. These modules automate data analysis, generate reports, and integrate external tools into Autopsy's platform. Python modules are used to automatically parse specific file types or search for custom patterns in the recovered data.

The study explores specific case studies that illustrate Python's impact on the speed, accuracy of forensic analysis, and how it improves the efficiency and effectiveness of forensic investigations.

Case Study 1: Automating File Hash Extraction for Malware Analysis (Fig.1)

```python
from org.sleuthkit.autopsy.ingest import IngestModule, IngestModuleFactoryAdapter
from org.sleuthkit.datamodel import SleuthkitCase, ReadContentInputStream, AbstractFile
import hashlib

# Known malicious hashes (example)
malicious_hashes = {
    "e99a18c428cb38d5f260853678922e03",  # Example MD5 hash
    "098f6bcd4621d373cade4e832627b4f6"
}

class HashCheckModule(IngestModule):
    def process(self, file):
        if file.isFile():
            # Compute MD5 hash
            md5_hash = hashlib.md5()
            input_stream = ReadContentInputStream(file)
            buffer = bytearray(1024)
            read = input_stream.read(buffer)
            while read > 0:
                md5_hash.update(buffer[:read])
                read = input_stream.read(buffer)

            file_hash = md5_hash.hexdigest()

            # Check if hash matches known malware
            if file_hash in malicious_hashes:
                self.logMessage(f"Malware detected: {file.getName()} ({file_hash})")

        return IngestModule.ProcessResult.OK
```

Fig. 1 *File Hash Extraction*[Грешка! Източникът на препратката не е намерен.].

Scenario: A disk image is scanned for known malware files by computing their hashes and comparing them against a malware database.

Solution: Python is used with Autopsy's API to extract file hashes from an image and compare them with a list of known malicious hashes.

The script reads each file, computes its MD5 hash, and checks it against a list of malicious hashes. If a match is found, an alert is logged. This helps in automating malware detection in forensic disk images. The investigator gets a report listing any detected malware based on hash comparison. It reduces manual effort in scanning for known malware files.

Case Study 2: Extracting Deleted Files for Investigation (Fig.2)

```python
from org.sleuthkit.autopsy.ingest import IngestModule
from org.sleuthkit.datamodel import SleuthkitCase, AbstractFile

class RecoverDeletedFilesModule(IngestModule):
    def process(self, file):
        if file.isFile() and file.isDeleted():
            self.logMessage(f"Deleted File Found: {file.getName()} at {file.getParentPath()}")
        return IngestModule.ProcessResult.OK
```

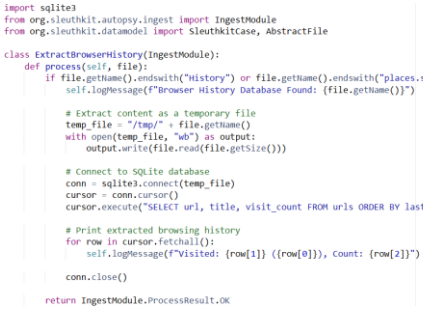
Fig. 2 *Extracting Deleted Files.*

Scenario: An investigator wants to recover deleted files from a disk image and analyze their contents.

Solution: Autopsy's API is used to search for files marked as deleted and extract them for analysis.

The script iterates through files in the disk image. It checks if the file is marked as deleted and logs its path. Investigators use this to restore deleted files for further examination. The forensic team gets a list of deleted files that may be relevant to the case. It helps in recovering evidence from formatted or tampered drives.

Case Study 3: Extracting Browser History for User Activity Analysis (Fig.3)



```
import sqlite3
from org.sleuthkit.autopsy.ingest import IngestModule
from org.sleuthkit.datamodel import SleuthkitCase, AbstractFile

class ExtractBrowserHistory(IngestModule):
    def process(self, file):
        if file.getName().endswith("History") or file.getName().endswith("places.sqlite"):
            self.logMessage(f"Browser History Database Found: {file.getName()}")

            # Extract content as a temporary file
            temp_file = "/tmp/" + file.getName()
            with open(temp_file, "wb") as output:
                output.write(file.read(file.getSize()))

            # Connect to SQLite database
            conn = sqlite3.connect(temp_file)
            cursor = conn.cursor()
            cursor.execute("SELECT url, title, visit_count FROM urls ORDER BY last_visit_time DESC")

            # Print extracted browsing history
            for row in cursor.fetchall():
                self.logMessage(f"Visited: {row[1]} ({row[0]}), Count: {row[2]}")

            conn.close()

        return IngestModule.ProcessResult.OK
```

***Fig. 3** Extracting Browser History.*

Scenario: An investigator tracks user activity by extracting browser history from a forensic disk image.

Solution: Python is used with Autopsy to locate browser history databases and extract URLs.

The script scans for browser history databases (Chrome's History or Firefox's places.sqlite). It extracts URLs, titles, and visit counts from the database. Investigators uses this data to track user activity. It provides a list of visited websites, helping in cases of cybercrime, insider threats, or evidence collection [3, 4, 5].

Case Study 4: Extracting Email Artifacts for Investigation (Fig. 4)

```
from org.sleuthkit.autopsy.ingest import IngestModule
from org.sleuthkit.datamodel import SleuthkitCase, AbstractFile
import email

class ExtractEmailsModule(IngestModule):
    def process(self, file):
        if file.getName().endswith(".eml"):
            self.logMessage(f"Email Found: {file.getName()}")

            # Read email file
            raw_email = file.read(file.getSize()).decode(errors="ignore")
            msg = email.message_from_string(raw_email)

            # Extract email headers
            sender = msg["From"]
            receiver = msg["To"]
            subject = msg["Subject"]
            date = msg["Date"]

            self.logMessage(f"From: {sender}, To: {receiver}, Subject: {subject}, Date: {date}")

        return IngestModule.ProcessResult.OK
```

***Fig. 4** Extracting Email Artifacts.*

Scenario: An investigator analyzes email communications stored on a suspect's device.

Solution: Autopsy is used with Python to extract MBOX/EML email files and parse their contents.

The script scans for email files (.eml format). It extracts headers such as sender, receiver, subject, and date. Investigators use this to analyze email communications. It extracts important evidence from emails that might contain fraud, phishing attempts, or insider threats.

Case Study 5: Extracting Metadata from Images for Digital Evidence (Fig. 5)

```
from org.sleuthkit.autopsy.ingest import IngestModule
from org.sleuthkit.datamodel import AbstractFile
from PIL import Image
from PIL.ExifTags import TAGS

class ExtractImageMetadata(IngestModule):
    def process(self, file):
        if file.getName().lower().endswith((".jpg", ".jpeg", ".png")):
            self.logMessage(f"Image Found: {file.getName()}")

            # Extract content as a temporary file
            temp_file = "/tmp/" + file.getName()
            with open(temp_file, "wb") as output:
                output.write(file.read(file.getSize()))

            # Open image and extract EXIF metadata
            img = Image.open(temp_file)
            exif_data = img._getexif()

            if exif_data:
                for tag, value in exif_data.items():
                    tag_name = TAGS.get(tag, tag)
                    self.logMessage(f"{tag_name}: {value}")

        return IngestModule.ProcessResult.OK
```

***Fig. 5** Extracting Metadata from Images.*

Scenario: An investigator extracts metadata (such as GPS coordinates) from images found on a suspect's device.

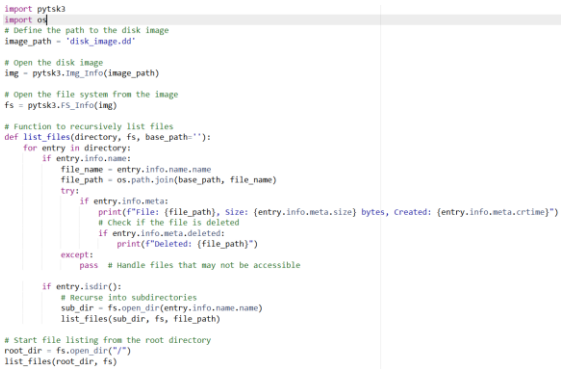
Solution: Python is used with Autopsy to extract EXIF metadata from images.

It extracts metadata from images, such as GPS coordinates, timestamps, and camera model and helps determine where and when an image was taken.

These Autopsy case studies show how Python automates digital forensics tasks such as file hash extraction, deleted file recovery, browser history analysis, email investigation, and metadata extraction. By using Python scripting, forensic investigators can save time, increase efficiency, and uncover crucial evidence faster.

Next case studies demonstrate how Python is used in conjunction with The Sleuth Kit (TSK) and Autopsy to enhance digital forensic investigations. Each case study is focused on solving real-world forensic challenges and highlights how Python scripts can automate and extend the capabilities of these tools.

Case Study 6: Automating Evidence Extraction with TSK and Python (Fig. 6)



```
import pytsk3
import os
# Define the path to the disk image
image_path = 'disk_image.dd'

# Open the disk image
img = pytsk3.Img_Info(image_path)

# Open the file system from the image
fs = pytsk3.FS_Info(img)

# Function to recursively list files
def list_files(directory, fs, base_path=''):
    for entry in directory:
        if entry.info.name:
            file_name = entry.info.name.name
            file_path = os.path.join(base_path, file_name)
            try:
                if entry.info.meta:
                    print(f"File: {file_path}, Size: {entry.info.meta.size} bytes, Created: {entry.info.meta.crtime}")
                    # Check if the file is deleted
                    if entry.info.meta.deleted:
                        print(f"Deleted: {file_path}")
            except:
                pass  # Handle files that may not be accessible

        if entry.isdir():
            # Recurse into subdirectories
            sub_dir = fs.open_dir(entry.info.name.name)
            list_files(sub_dir, fs, file_path)

# Start file listing from the root directory
root_dir = fs.open_dir("/")
list_files(root_dir, fs)
```

***Fig. 6** Automating Evidence Extraction.*

Scenario: The forensic task is with analyzing a disk image to recover files and identify any deleted files from a suspect's computer. The disk image has multiple partitions, and the forensic specialist recovers both live and deleted files by automating the extraction of relevant file metadata (e.g., file names, timestamps, file sizes) and deleted files to save time.

Solution: Using Python bindings for The Sleuth Kit (PyTSK), a Python script is developed to automate the analysis process. The script (Fig. 6) accesses the disk image and examine the file system, extract file metadata for all files, including those that are deleted, and generate a report that summarizes the findings.

The script uses PyTSK to open the disk image and access the file system. It recursively lists all files, extracting metadata (file name, size, creation time) for each file, including deleted files. The script identifies deleted files by checking the deleted attribute in the file's metadata. The script automates the process of recovering metadata and deleted files, significantly reducing the time spent on manual analysis.

Case Study 7: Hash Comparison for Malware Detection in Autopsy (Fig. 7)

```python
import hashlib
import os
from autopsy.lib import Autopsy
# Define the known malware hash database (e.g., SHA256 hashes of known malware files)
malware_hash_db = set([
    'd41d8cd98f00b204e9800998ecf8427e',  # Example hash (replace with actual)
    'e99a18c428cb38d5f260853678922e03'
])
# Function to calculate SHA256 hash of a file
def calculate_hash(file_path):
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        # Read file in chunks of 4K
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)
    return sha256_hash.hexdigest()
# Function to compare hashes with the malware database
def check_for_malware(file_path):
    file_hash = calculate_hash(file_path)
    if file_hash in malware_hash_db:
        return True, file_hash  # Return True if malware hash is found
    return False, None
# Integration with Autopsy (e.g., scanning all files in the disk image)
def scan_disk_image_for_malware(disk_image_path):
    # Open the disk image in Autopsy and iterate through files
    for root, dirs, files in os.walk(disk_image_path):
        for file_name in files:
            file_path = os.path.join(root, file_name)
            is_malware, file_hash = check_for_malware(file_path)
            if is_malware:
                print(f"Malware Detected: {file_name}, Hash: {file_hash}")
                # Flag or mark the file as suspicious in Autopsy
                # Autopsy provides an API to flag files within its interface
                Autopsy.add_flag(file_path, "Malware Detected")
# Run the malware scanning function
scan_disk_image_for_malware('path_to_disk_image')
```

***Fig. 7** Hash Comparison for Malware Detection.*

Scenario: The forensic task is with analyzing a disk image to detect potential malware. The forensic specialist compares the hashes of files in the disk image against a known database of malware hashes. This process is prolonged if done manually, especially for large disk images.

Solution: A custom Autopsy module is developed in Python to automate the process of generating file hashes and comparing them with a known hash database of malware signatures.

The script calculates the SHA256 hash for each file in the disk image. It compares the calculated hash against a pre-existing database of known malware hashes. If a match is found, the script flags the file as potentially malicious, which can be reported within Autopsy. The script automates the comparison of file hashes with known malware signatures, speeding up the detection of malicious files. The integration with Autopsy allows to visualize and flag the suspicious files within the Autopsy interface, enabling quick follow-up analysis.

These case studies demonstrate the power of Python in enhancing the capabilities of The Sleuth Kit (TSK) and Autopsy. By automating repetitive tasks, recovering deleted files, detecting malware, and generating reports, Python significantly improves the efficiency, accuracy, and speed of digital forensic investigations. These practical examples show how Python can be applied to real-world challenges in digital forensics, from simple file metadata extraction to comprehensive forensic workflows.

Next case studies demonstrate how Python can be used in conjunction with Volatility for various memory forensics tasks, such as analyzing processes, detecting network connections, identifying suspicious DLLs, and uncovering rootkit activity. By automating these processes, Python enhances the forensic analysis, making it faster, more efficient, and less error-prone.

Below are practical case studies demonstrating how to use Python with Volatility, the popular memory forensics tool, to analyze memory dumps. Each case study includes a Python code snippet for specific tasks in memory analysis, focusing on different forensic needs.

Case Study 9: Analyzing Process Information with Volatility

Scenario: The goal is to analyze a memory dump to extract detailed information about the running processes at the time the memory image was captured. This identifies suspicious processes that are part of a malware attack or unauthorized activities.

Solution: Using Volatility with Python, the detailed information about the processes, including process ID, parent process ID, executable name, and the memory usage is extracted and then printed.

The Volatility framework is used to extract the list of processes in the memory dump. In this case, the linux.pslist plugin is applied to the memory dump of a Linux-based system. The script runs the volatility3 command-line tool using Python's os.popen() to interact with Volatility, extract the process information, and display it. The script outputs a list of processes, displaying information such as Process ID (PID), Parent PID (PPID), executable name, and the associated memory usage. The output to identify any unusual or suspicious processes running at the time the memory dump is examined and captured.

Case Study 10: Extracting Network Connections from a Memory Dump

Scenario: In this case, network connections and socket information from a memory dump is extracted to determine if any unauthorized network activity or malware communication occurred during the system's runtime.

Solution: Volatility's netstat plugin (for older versions of Volatility, or adapt plugins in Volatility 3) is used to extract network connections, including details such as IP addresses, ports, and the associated processes.

The script utilizes volatility3 to invoke the linux.netstat plugin, which extracts information about the system's active network connections. By using Python's os.popen() to execute the command, the output is captured and printed to the console. The script returns a list of network connections, showing information such as local and remote IP addresses, port numbers, and the associated processes. This information is used to identify potentially malicious network activity or unauthorized data exfiltration.

Case Study 11: Detecting Suspicious DLLs in Memory (Windows)

Scenario: The objective is to analyze a memory dump from a Windows system to detect any suspicious or injected DLLs. Malicious processes may inject DLLs into legitimate processes, making it important to examine the memory for such anomalies.

Solution: Volatility is used to list loaded DLLs in the memory image and then cross-reference them with known malicious DLL hashes or a whitelist.

The script first lists the loaded DLLs in a memory dump using the windows.dlllist Volatility plugin. It then calculates the hash of each DLL and compares it to a list of known malicious DLL hashes. If any of the DLLs' hashes match known malicious ones, they are flagged as suspicious. The script outputs the list of loaded DLLs in the memory dump. If any DLL matches the predefined malicious hash database, it is flagged as suspicious and printed for further investigation.

Case Study 12: Detecting Rootkits Using Volatility (Windows)

Scenario: The presence of a rootkit that might have hidden processes or files is detected. Rootkits often operate by manipulating the operating system to hide their presence.

Solution: Volatility is used to look for hidden processes or modules that are indicative of rootkit behavior. This is achieved using Volatility's pslist, pstree, and dlllist plugins.

The script runs the pslist and pstree plugins to list all processes and show the process tree. Rootkits often hide processes by tampering with the process list, so discrepancies between these outputs can be indicative of hidden or malicious activity. The dlllist plugin is used to list the loaded modules (drivers), which can help identify any hidden or malicious drivers loaded by rootkits. The output reveals any hidden processes or drivers, which may indicate the presence of a rootkit on the system. If suspicious processes or hidden drivers are detected, they can be further investigated for rootkit activity [6,7,8,9].

Below are practical case studies demonstrating how to use Scapy, a powerful Python library for network packet manipulation

and analysis. Each case study includes Python code to help with real-world forensic and security investigations.

Case Study 13: Network Traffic Sniffing and Analysis

Scenario: A cybersecurity analyst suspects unauthorized data exfiltration from a corporate network. They need to monitor network traffic to detect suspicious activity.

Solution: Use Scapy to sniff network packets, filter HTTP traffic, and identify potential data leaks.

The script captures IP packets and prints the source and destination IP addresses. It filters only IP traffic, but it can be modified to capture specific protocols like TCP, UDP, or HTTP. This is useful for detecting unauthorized connections or data transfers. The captured packets are reviewed to identify suspicious connections. Unusual outbound connections indicate data exfiltration.

Case Study 14: ARP Spoofing Detection

Scenario: An IT security team suspects an ARP spoofing attack on the network, where an attacker is trying to intercept traffic.

Solution: Scapy is used to detect ARP anomalies, such as multiple MAC addresses responding to the same IP.

The script monitors ARP replies and maintains an IP-to-MAC address mapping. If an IP is seen with a different MAC than before, it flags a potential ARP spoofing attack. This helps security teams detect MITM (Man-in-the-Middle) attacks. If an attacker is trying to spoof ARP responses, it will be detected and alerted. The security team can take countermeasures, such as blocking the attacker's MAC address.

Case Study 15: Port Scanning Detection

Scenario: A security analyst wants to identify if someone is scanning the network for open ports, a common reconnaissance technique used by attackers.

Solution: Scapy is used to detect repeated connection attempts to multiple ports.

The script monitors TCP traffic and tracks how many different ports each source IP is probing. If an IP scans more than 5 ports, it raises an alert. This is useful for detecting reconnaissance activity from attackers. If a hacker is scanning the network for vulnerabilities, the system administrator is alerted. Security teams block the attacker's IP or implement firewalls.

Case Study 16: Crafting and Sending Custom Packets

Scenario: A penetration tester sends a custom crafted SYN packet to test a firewall rule.

Solution: Scapy is used to create and send a TCP SYN packet to a specific target.

The script crafts a TCP SYN packet (used to initiate a connection) to a specified IP and port. This is useful for penetration testing or testing firewall rules. If the firewall is properly configured, it should block the request. The tester can verify if the firewall blocks SYN packets or allows unauthorized access. If the packet gets through, security policies need to be updated.

Case Study 17: DNS Spoofing Detection

Scenario: A network administrator suspects that a DNS spoofing attack is redirecting users to malicious websites.

Solution: Scapy is used to monitor DNS responses and check for inconsistencies.

The script captures DNS responses and checks for changes in IP resolution. If the same domain resolves to different IPs over time, it indicates DNS spoofing. This helps prevent phishing attacks and man-in-the-middle (MITM) attacks. The script flags potential DNS spoofing, allowing security teams to investigate further. Users avoid being redirected to malicious websites.

These case studies demonstrate how Scapy is used in cybersecurity investigations and forensic analysis. From sniffing traffic and detecting attacks to crafting custom packets, Scapy is a versatile tool for network security professionals.

3. Conclusion

These case studies show how Python can automate digital forensics tasks such as file hash extraction, deleted file recovery, browser history analysis, email investigation, and metadata extraction. By using Python scripting, forensic investigators can save time, increase efficiency, and uncover crucial evidence faster. The benefits of using Python with computer forensic tools are automation, customizability, extensibility, interoperability, and integrating different forensic tools into a single workflow.

The report is in implementation of the National Scientific Program "Security and Defense", adopted with RMS No. 731/21.10.2021, and financed by the Ministry of Education and Science of the Republic of Bulgaria according to Agreement No. D01-74/19.05.2022.

4. References

1. Kamble, Dhwaniket and Nilakshi Jain. “DIGITAL FORENSIC TOOLS: A COMPARATIVE APPROACH.” (2015).
2. Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., & Zhang, Y. (2024). A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. High-Confidence Computing, 100211.
3. V. Petrova, "The Hierarchical Decision Model of cybersecurity risk assessment," 2021 12th National Conference with International Participation (ELECTRONICA), Sofia, Bulgaria, 2021, pp. 1-4, doi: 10.1109/ELECTRONICA52725.2021.9513722.
4. V. Petrova, Using the Analytic Hierarchy Process for LMS selection, CompSysTech '19: 20th International Conference on Computer Systems and Technologies, June 2019, Ruse, Bulgaria, Pages 332–336, ISBN: 978-1-4503-7149-0.
5. V. Petrova. A decision hierarchical model of cyber security risk assessment. Mathematics and Education in Mathematics, 2021, 50: 191-195.
6. M. Sotirov, V. Petrova and D. Nikolova-Sotirova, D. Exploring the Impact of Educational Serious Game in a Gamified LMS. In 2024 International Conference Automatics and Informatics (ICAI) (pp. 307-312). IEEE.
7. M. Sotirov, V. Petrova and D. Nikolova-Sotirova, "Learning through Gamification: A Case Study on the Development and Integration of a University Educational Serious Game," 2024 International Conference Automatics and Informatics (ICAI), Varna, Bulgaria, 2024, pp. 313-318, doi: 10.1109/ICAI63388.2024.10851688.
8. M. Sotirov, V. Petrova and D. Nikolova-Sotirova, "Evaluation approach for the impact of a Gamified University Environment," 2024 8th International Symposium on Innovative Approaches in Smart Technologies (ISAS), İstanbul, Turkiye, 2024, pp. 1-6, doi: 10.1109/ISAS64331.2024.10845706.
9. O. I. Zhelezov and V. M. Petrova, "An Accelerated SFTessellation Algorithm for Obtaining a Voronoi Diagram," 2024 23rd International Symposium on Electrical Apparatus and Technologies (SIELA), Bourgas, Bulgaria, 2024, pp. 1-4, doi: 10.1109/SIELA61056.2024.10637861.