WORKLOG.

BY

AKINTUNDE AKINRO – 300389708

1 WORK DATE / HOURS LOG

F25_4440_G12

Date	Number of Hours	Description of Work
October 25th	5 hours	Research on all tools used to have clear understanding of planned approach and how to implement novel tool for project. Telegram installation on the emulator was verified using pm list packages | grep telegram, and the application directory was confirmed at /data/data/org.telegram.messenger. The app's data was then pulled to the emulator's SDCard using adb pull and copied to the PC extraction folder. Andriller was used with **Extract Shared Storage** and **App Data Decoders**, producing backup files (backup.ab), data/ and shared/ folders, and HTML/Excel reports. **For the novel tool attempt,** I Chose tkinter for GUI framework (built-in, no external dependencies). Selected subprocess module for ADB command execution. Designed file structure with modular approach (main.py + modules/ directory). Researched all requirements for dependency management (minimal dependencies for portability).
October 26th	3.5 hours	Developed custom Python script to automate extraction from Andriller outputs. The script scanned SQLite databases in the data/ folder, exported messages, chats, dialogs, and user data to CSV, extracted media from shared/, and identified potential keys in shared_prefs/. Path handling issues were resolved using pathlib.Path. **For the novel tool attempt**, I Added device listing functionality with `list_devices()` method that parses `adb devices` output and extracts device IDs. Implemented robust error handling for connection failures, device disconnections, and command timeouts. Added support for device-specific command execution using `-s` flag for multi-device scenarios. Had multiple errors and spent a load of time debugging all errors and improving error handling.

October 27th	3 hours	Researched on articles relating to andriller to gain extensive knowledge of the tool and how to use it and what can be potentially implemented in our novel tool. Looked into python as a language to use due to observation of other tools and libraries they used. Set up ALEAPP and ran on the Andriller extraction folder after installing dependencies. The tool generated HTML dashboards, CSV logs, and a _laval_artifacts folder summarizing app and device artifacts. This confirmed successful parsing of messages, chats, contacts, and media, while secret chats remained encrypted as expected. Local MTProto keys were successfully retrieved from shared_prefs and database files. **For the novel tool attempt**, Implemented comprehensive device information gathering with `get_device_info()` method. Extracted device model using `ro.product.model` property. Retrieved Android version via `ro.build.version.release`
October 28th	3 hours	**For the novel tool attempt**, Attempted to Enable Root but couldn't make it work. I tried using the standard adb root command, which consistently returned the "adbd cannot run as root" error due to unsupported system images. Second, I attempted to elevate privileges using adb shell su -c id, but the emulator lacked a working su binary, causing every privilege-escalation test to fail. Third, I tried enabling root by remounting the system partition with adb remount, but this also failed because the device was running in secure mode and blocked write access.
October 29th	2 hour	**For the novel tool attempt**, Finished information gathering. Captured SDK version through `ro.build.version.sdk`. Obtained manufacturer information from `ro.product.manufacturer`. Retrieved device name from `ro.product.device`. Extracted serial number using `get-serialno` command.
October 30th	2 hours	Research on novel tool implementation.
November 2nd	3.5 hours	**For the novel tool attempt**. Implemented root status checking with multiple verification methods. Created JSON-formatted output for easy parsing and display. Added error handling for missing properties and connection issues.
November 3rd	2.5 hours	
November 5th	2 hours	
November 6th	1 hours	**For the novel tool attempt**, tried to fix Enabled Root functionality on the app. I first tried adb root and ADB server restarts, but the tool still showed no root. I then attempted UID verification using adb shell su -c "id", echo tests, and whoami checks, all of which failed to trigger detection in the tool. Each method confirmed root

		on the emulator, but the Python ADB module required further adjustments to recognize elevated permissions.
November 8th	2 hours	
November 11th	3.5 hour	**For the novel tool attempt**, Created `enable_root()` method that executes `adb root` command, handles ADB server restart (kill-server and start-server), implements reconnection logic with device verification, and provides detailed error messages for troubleshooting. Implemented `disable_root()` method for security purposes. Added `is_emulator()` detection using `ro.kernel.qemu` property and `ro.hardware` checks
November 12th	3 hours	
November 14th	4 hours	**For the novel tool attempt**, tried implementing logcat log extraction with package-specific filtering. Created `extract_logs()` method that executes `logcat -d | grep {package}` to capture app-specific logs. Implemented log file writing with UTF-8 encoding for proper character handling. Created `logs/` subdirectory for log files. Added timestamp and package information to log files. Implemented error handling for logcat command failures. Not yet working. Still in progress.
November 15th	2 hours	

2 SUMMARY OF WORK DONE

The Novel Forensic Tool for 4440 is an ongoing project built in Python with a tkinter-based GUI, where we attempt to create a tool that combines key functionalities from Andriller (for data extraction), MVT (for monitoring), and MobSF (for security analysis) into a single cohesive platform. We have made strong progress, successfully addressing two major roadblocks: enabling root access on various emulator system images and achieving consistent emulator detection through the ADB connector. These challenges were overcome using multiple root-verification techniques, enhanced error handling, and improved device-recognition logic. With these issues resolved, the tool has now entered the extraction phase, where we are attempting to successfully retrieving Telegram data including messages, media files, call logs, and login events, as well as other app-related data from accessible directories.

Root functionality was implemented using five independent verification methods—ADBD status checks, UID verification (adb shell id), SU command testing, echo validation, and whoami confirmation alongside automated ADB server restart sequences with short delays to ensure reliable connection and root access across emulators. This architecture provides a stable foundation for real-time monitoring, structured data extraction, and security analysis, and will support further development toward advanced features like database parsing, network capture, event correlation, and comprehensive forensic reporting.

For extracting Telegram data, I focused on **Andriller** for direct data extraction and **MVT** for monitoring and forensic indicators. Using Andriller, I pulled the /data/data/org.telegram.messenger folder from the emulator, successfully retrieving key components such as application databases, message logs, media files, and shared preferences. This provided the foundation for in-depth analysis, despite initial challenges with limited root access on the emulator.

To process the extracted data, I developed a **Python script** that automated parsing of Andriller outputs. The script scanned the data/ folder for SQLite databases, exporting messages, chats, dialogs, and user information into structured CSV files. Media files from the shared/ folder were extracted into organized directories, and the script also analyzed shared_prefs to locate possible encryption or authentication keys. This automation streamlined the process, ensured consistency, and prepared the data for further analysis.

In parallel, **MVT** was employed to monitor Telegram activity and verify forensic indicators. This included tracking login events, OTPs, and other application behaviors, complementing the extraction with real-time data about app activity. The combination of extraction via Andriller and monitoring via MVT allowed for a more complete forensic snapshot of Telegram usage on the emulator.

To extend the analysis further, the **ALEAPP tool** was applied to the Andriller extraction outputs. ALEAPP parsed the extracted files and generated HTML dashboards, CSV logs, and a _laval_artifacts folder summarizing app and device artifacts. This enabled visualization of messages, media, user accounts, and events, providing a higher-level overview that supported both detailed examination and reporting. Secret chats remained encrypted as expected, but local MTProto keys were successfully retrieved from shared_prefs and database files, confirming the integrity of the extraction process.

Overall, this workflow combining Andriller extraction, Python-based parsing, MVT monitoring, and ALEAPP analysis produced a comprehensive dataset. Messages, chats, contacts, media, and potential encryption keys were all successfully captured, structured, and organized for reporting. This multi-tool approach not only validated the effectiveness of the extraction process but also provided a solid foundation for subsequent forensic reporting and deeper analysis of mobile application behavior

3 AI USE SECTION

AI Tool used	Version, AccountType	Specific Feature used for
GPT-5	5.1 codex mini	Used to understand the basic implementation steps needed to create a novel tool for forensics
Claude	Sonnet 4.5 and Haiku 4.5	Used to correct code errors and fix bugs and provide error handling.

4 VALUE ADDITION

During the development of the Forensic Tool for 4440, I leveraged AI-assisted coding tools to address persistent root detection issues in the emulator. Although root privileges were present on the device, the tool was unable to detect them through the Python ADB module. I experimented with multiple approaches suggested by AI, including ADB root commands, UID verification via adb shell su -c "id", echo-based checks, and whoami validations, iteratively refining the code to update the GUI dynamically. AI also helped me design robust retry logic and diagnostic buttons, ensuring that root status could be refreshed or re-evaluated without restarting the entire application. Additionally, AI guidance contributed to modularizing the enable/disable root functionality, improving the user interface, and integrating error handling so the tool could reliably respond to different emulator configurations.

5 APPENDIX

Prompt: "Write a Python function to check if root is enabled on an Android emulator using ADB and return True or False."
AI Response / Suggestion:

```
import subprocess


def check_root():
    try:
        result = subprocess.run(['adb', 'shell', 'su', '-c', 'id'],
                                capture_output=True, text=True, timeout=5)
        if 'uid=0' in result.stdout:
            return True
        return False
    except Exception:
        return False
```

I tested it, but it returned inconsistent results on some emulator images.

Prompt: "Update a Tkinter label in real-time with root status after checking with ADB."
AI Response / Suggestion:

```
self.root_status_var = tk.StringVar()

self.root_status_label = tk.Label(root_frame, textvariable=self.root_status_var)

self.root_status_var.set("Root Status: Checking...")


def refresh_root_label():

    is_rooted = check_root()

    self.root_status_var.set("Root Status: Enabled" if is_rooted else "Root Status: Disabled")
```

I integrated this with threading to prevent GUI freezing during ADB calls.

My emulator has root enabled, but my Python tool doesn't detect it. How can I check root properly using ADB?"

You can combine several ADB-based checks because sometimes adb shell su -c "id" fails on certain images. Here's an approach using multiple commands:

```
def check_root():

    commands = [

        "adb shell su -c 'id'",

        "adb shell whoami",

        "adb shell 'echo ROOT_CHECK'",

        "adb shell getprop ro.secure"

    ]

    for cmd in commands:

        try:

            result = subprocess.run(cmd.split(), capture_output=True, text=True, timeout=3)

            if "0" in result.stdout or "root" in result.stdout:
```

```
        return True
    except:
        continue
return False
```

I recommended integrating all these checks into one function to improve reliability. You ran it but initially got inconsistent results on different emulator images.