ENSF 609 & 610 Final Report

Ransomware Detection on Linux

Due Date:

April 11, 2025

Industry Partner:

Louis Lee, CISCO Systems

Academic Advisor:

Dr. Yves Pauchard

Group Members:

Cole Cathcart

Destin Saba

Rana El Sadig

David Nguyen

Project Repository:

https://github.com/colecathcart/609Capstone

CONTENTS

1. INTRODUCTION

This report outlines our ENSF 609/610 capstone project sponsored by Cisco. The project goal was to develop and test a proof-of-concept ransomware detection and protection software for the Linux operating system. The solution, as described by the sponsor, should be a standalone, configurable program that can mitigate and remove ransomware threats while optimizing resource usage. We successfully created a robust software which can protect a system from multiple ransomware attack strategies, and refined our solution over multiple iterations. Conclusions about the efficacy of our chosen detection method can be found in Section 3.

1.1 Problem Background

Ransomware is a specialized type of malware that has grown in popularity in recent years. Unlike other forms of malware which typically attempt to destroy systems or steal sensitive information, ransomware is unique in that its primary goal is to hold a device or system "hostage," undamaged but also unusable until the user pays a ransom fee to the attacker. This type of malware has been at the forefront of security concerns since 2017, when a massive outbreak of the WannaCry ransomware infected hundreds of thousands of devices. Despite increased scrutiny since then, ransomware attacks continue to cost individuals and corporations enormous sums of money annually.

As a leader in device security, Cisco Systems is invested in potential solutions to ransomware. One area they would like to see progress in is the development of a ransomware detector specific to Linux machines. Although ransomware more often targets Windows and Mac OS due to the large casual user base, attacks against Linux are becoming more common due to the widespread use of Linux for server computers. Servers are a prime target for ransomware, as they are usually operated by corporations who can afford to pay out much larger sums than individuals. To aid in their research, our team is developing a standalone linux ransomware detector that will serve as a proof of concept for possible implementations by Cisco in the future.

1.2 Tech Stack

The tech stack used for this project is as follows:

- **C++**: The majority of detector code is written in C++. We chose this language because of its fast execution time, its ability to take on an object-oriented structure, and its wide support for linux system interactions. We used several external libraries to interact with the linux filesystem and extend the functionality of C++:
    - **fanotify** for monitoring events in the filesystem.
    - **openSSL** for creating hash checksums and other cryptographic purposes.
    - **libmagic** for file type checking.
    - **mqueue** for inter-process communication.

    - **Qt** for GUI development.
    - **WebSocket++** for two-way communication between client detectors and the central dashboard.
    - **SQLite** as a lightweight embedded database to store system metrics, device statuses, and known malicious hash signatures.
    - **gtest** for unit testing.
- **Python**: Our sample ransomware, GonnaCry, is written in Python.
- **Ubuntu**: Our solution was developed on Ubuntu 24.04 LTS. We chose this distro because of its quick setup and wealth of online resources. We used **VMWare** virtual machines to run linux in a clean environment that could be easily restored in the case of corruption.
- **ReactJS:** The administrative dashboard was built as a single-page application using React
- **Other technologies**: We used **git** and **Github** for version control tracking.

2. PROJECT OVERVIEW

2.1 Ransomware Emulator: GonnaCry

2.1.1 GonnaCry Overview

To ensure the practicality and effectiveness of our detector, we found a realistic sample ransomware to test with. GonnaCry is a ransomware emulator inspired by the infamous WannaCry ransomware from the worldwide attack in 2017 [1]. Its original purpose was to demonstrate common ransomware techniques, particularly focusing on file encryption, key management, and persistence mechanisms used in real-world malware. GonnaCry utilizes a hybrid encryption scheme, combining symmetric encryption (AES-256-CBC) to encrypt victim files, and asymmetric encryption (RSA-2048) to securely encrypt the AES keys. This hybrid encryption scheme is one of the most commonly employed schemes by modern ransomware, as demonstrated in figure 1. The original implementation included features such as network-based key exchange, system persistence via cron jobs and systemd services, and altering the victim's desktop wallpaper to display a ransom demand.

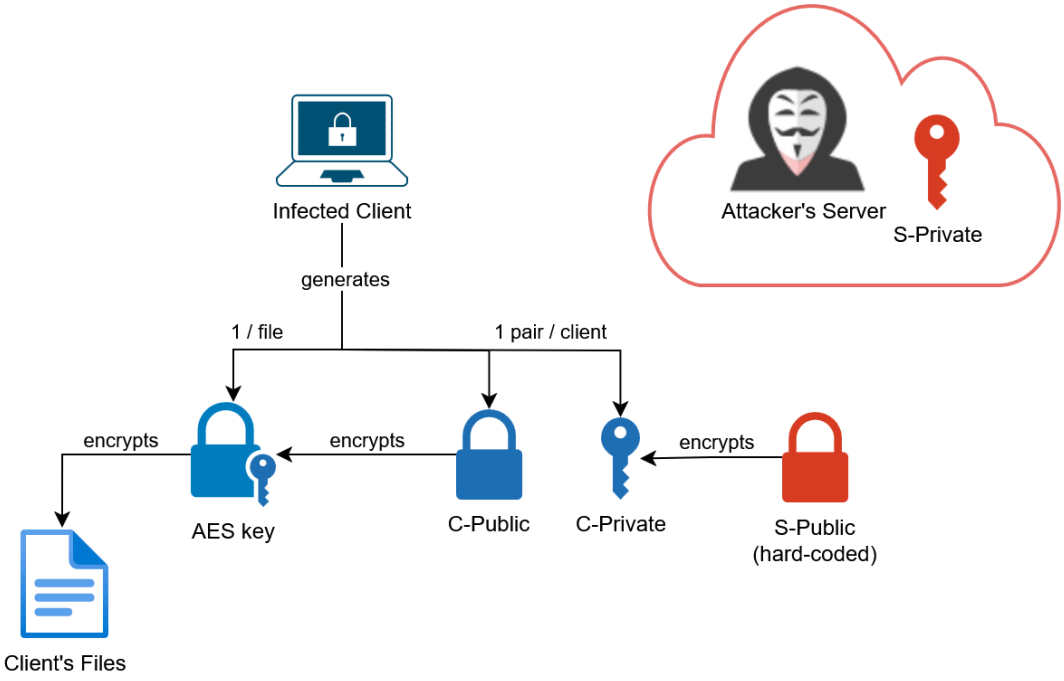


Figure 1. Hybrid encryption scheme employed by GonnaCry, WannaCry, and other common ransomwares

2.1.2 Modifications to GonnaCry

For the purpose of safe and controlled ransomware detection testing, our project made several critical modifications to the original GonnaCry codebase:

- **Containment of file encryption:**
  Encryption and decryption processes were strictly limited to a designated test directory, ensuring no unintended encryption of critical system or personal files.
- **Elimination of external dependencies:**
  All external server dependencies were removed. Instead, encryption keys (RSA and AES) are generated and stored locally within the test environment. All communication with the server to exchange keys for decryption was simulated.
- **Disabling harmful persistence mechanisms:**
  Potentially dangerous functionality such as deleting system snapshots, system persistence through cron jobs, background daemons, and process termination were either completely disabled or safely simulated to avoid any unintended system alterations during tests.
- **Fixing outdated code and packages:**
  Outdated packages that are no longer supported were replaced, such as pycrypto with pycryptodome.

These adaptations allowed GonnaCry to serve effectively as a realistic but controlled testbed for assessing and improving our ransomware detection strategies.

2.1.3 Initial Testing Results: First Iteration Detector Evaluation

To evaluate the effectiveness of our ransomware detection system, we conducted initial testing using the adapted GonnaCry emulator on our first iteration of the detector after the midterm evaluation. The first iteration of our detector relied heavily on measuring file entropy under the assumption that files encrypted by ransomware using strong encryption methods, such as AES-256, would exhibit significantly higher entropy compared to their unencrypted counterparts. Therefore, we expected that GonnaCry's AES encryption process would cause encrypted files to surpass the established entropy threshold, triggering a successful detection.

However, the results of this initial testing revealed an unexpected outcome. Despite utilizing strong AES-256 encryption, GonnaCry successfully evaded our detector, completely bypassing entropy-based analysis. This result was surprising and led us to further investigate the ransomware's internal mechanisms.

Upon closer examination, we discovered that GonnaCry incorporated an additional step after encrypting the file contents: it applied Base64 encoding to the encrypted data. Base64 encoding, while simple, significantly reduces the apparent entropy of the encrypted data, effectively disguising it as less random and more similar to ordinary, non-encrypted files. This technique, commonly used by real-world ransomware, is an effective strategy for evading entropy-based detection.

Our test clearly demonstrated that solely relying on entropy as a detection metric is insufficient, as simple encoding methods like Base64 can substantially mask the high-entropy signatures of encrypted files. These initial findings underscored the need for a multi-layered detection approach, prompting subsequent improvements in our ransomware detection methodology.

2.2 Ransomware Detector

What follows is a high-level description of the techniques used by our ransomware detector. To aid in understanding our system, two diagrams are included in the appendices: A simplified UML class diagram of our detector in Appendix B, and an event state diagram in Appendix C.

2.2.1 File System Monitoring

To rapidly detect and stop a ransomware attack in-progress, our system continuously listens on the filesystem using the *fanotify* library. This library produces an event whenever a read, write, delete, and other action is performed anywhere in the file space, thus ensuring that any encryptions made during a ransomware attack will be picked up by our system. Due to the

enormous number of events generated by the computer during normal operation, we apply several filters before events reach the analysis stage:

- We begin by ignoring all events that are not file writes. Since a file write is always required to encrypt a file, we only need to watch for these events
- Next, we ignore any event coming from a hidden directory (beginning with '.') or system directories which do not contain "valuable" files but are only used by the OS, such as /tmp. These can be ignored because ransomware is chiefly concerned with encrypting files created by the user and usually tries to avoid files which may be needed for normal system operation, as this would hamper the chances of a ransom being paid.
- Finally, we ignore any other directories specified in the allowlist (see Section 2.2.6)

The filtering stage significantly reduces the number of events on the system that are being fed into our analyzer, increasing the efficiency of our solution.

2.2.2 Entropy Checking

To detect whether a file write represents a potentially malicious encryption, we measure the entropy of the file's contents. Shannon entropy measures the uncertainty or randomness of a file's byte distribution. Higher entropy values suggest encrypted or highly compressed data, which can indicate potential ransomware activity. The entropy is calculated using the formula:

$$H(X) = -\sum p(x) \log_2 p(x)$$

where $p(x)$ represents the probability of each unique byte in the file. We classify files with an entropy greater than 7.5 as potentially encrypted and flag their associated processes for further monitoring or termination.

The detector uses small samples to calculate file entropy more efficiently. Instead of scanning entire files, the detector reads a 1MB segment to check for high entropy, and only increases the read size (up to 100MB) if the same process recurs. If a single high-entropy megabyte block is detected (entropy > 7.5), the process is immediately terminated. This approach significantly boosts efficiency and enhances the system's ability to detect "partial encryption" ransomware early in its execution.

Some files such as images, video, or compressed formats naturally have high entropy. To avoid false positives on these files, we employ a different technique called the NIST Monobit Frequency Test[2]. This test evaluates the balance of 1s and 0s in a file's binary representation. It is a more expensive operation than entropy analysis but can distinguish between unencrypted and encrypted files with much higher precision, even on naturally high-entropy file types. The test statistic is calculated as:

$$S_n = \left| \sum_{i=1}^{n} X_i \right|$$

where X_i represents individual bits. The observed value is then standardized as:

$$S_{obs} = \frac{S_n}{\sqrt{n}}$$

A p-value is determined using:

$$p = \text{erfc}\left(\frac{S_{obs}}{\sqrt{2}} \right)$$

In our code, we count a block as passing when its p-value is ≥ 0.01. If 90% or more of a file's blocks pass this test, we consider the entire file to be random, which in our detection logic is treated as suspicious and indicative of encryption.

By combining entropy analysis with the Monobit test, we enhance the accuracy of our detection system while reducing false positives for legitimate high-entropy files like compressed archives and images.

2.2.3 Magic Bytes

An improvement in the system involves determining file types based on their MIME type using "magic bytes" through the libmagic library, rather than relying on file extensions. This method significantly enhances security, as magic bytes are much harder to spoof, making it more reliable for identifying true file formats. Additionally, this approach helps reduce false positives by accurately distinguishing between genuinely high-entropy files (like .zips and images) and potentially malicious ones.

2.2.4 Multithreading

The system makes use of multithreading to optimize the processing of high-cost entropy calculations by enabling concurrent analysis of multiple files. A thread pool is used to analyze events produced by the event detector, following a producer-consumer pattern. This parallel approach significantly increases processing speed and efficiency, allowing for much faster detection and analysis of events.

2.2.5 Base64 Decoding

As encountered with the GonnaCry ransomware, applying base64 encoding is a common technique to lower file entropy after encryption. To defeat this technique, the detector checks each block for base64 encoding before calculating entropy or performing the monobit test. If the block is encoded, the detector will decode it before calculating entropy. This method enabled the detector to catch the GonnaCry ransomware, but it is still susceptible to other types of encoding (see section 2.5.2).

2.2.6 Configurable Allowlist and Denylist

To improve the usability of the detector, a configurable directory allowlist was added. This list contains directories that the ransomware detector will ignore file system events in. The primary use for this allowlist is to enable the user to perform purposeful encryption without the detector flagging the process as ransomware.

To improve detection speed in some cases, a configurable file extension denylist was also added. This list contains suspicious extensions that are typically associated with ransomware, such as .crypt, .enc, or .lock. If a process creates a file with an extension on this list, it is immediately flagged as suspicious and killed.

Both the directory allowlist and file extension denylist are easily updated through the GUI, and are currently maintained in separate .txt files.

2.2.6 Performance Analysis

Performance analysis shows that the system operates efficiently with a time complexity of O(n), where n is number of bytes processed or the maximum size used for entropy calculations, and a memory complexity of O(n), where n is the number of unique processes observed. The detector uses around 7.5MB of memory at startup and scales with the number of processes. CPU usage is minimal when idle but can spike when the detector is under heavy load, such as during large package install where thousands of file writes are performed. Exact metrics for CPU usage are hard to quantify, but one example tested on a VM with a 4-core Intel i7 processor (a middle-low end CPU), saw spikes up to 50-70%. While the peak load is high, the detector learns to trust a process after it has completed 30 unsuspicious writes and stops calculating entropy for further writes after that point. Therefore, the high CPU usage is always limited to a few seconds at most and will not result in a noticeable slowdown of computer operation.

2.2.7 Other Considerations

The system incorporates additional features to enhance flexibility and network-wide security. A singleton logger design allows redirection of message outputs based on command-line

arguments, enabling logs to be sent to a POSIX message queue for GUI integration, a separate text file, the terminal, or any combination of the three. Additionally, the system hashes detected ransomware executables, which could enable the sharing of these hashes across devices on a network or for use in a broader antivirus software. The overall design of the detector makes optimal use of data structures and algorithms to use as few system resources as possible during operation.

2.3 GUI

The Graphical User Interface (GUI) for the ransomware detector system serves as a user-friendly control panel for monitoring and managing the ransomware detection process. Designed using the Qt C++ library, the GUI operates independently from the detection engine, meaning it can function without the detector running and vice versa. It provides real-time updates on the detector's status, resource usage, and output logs, and allows users to start or stop the detector as well as select detection methods such as entropy-based analysis or honey files (note that honey files were not implemented in time for this report). The interface also enables users to manage allowlists and denylists through .txt files. Communication with the detector is achieved via a POSIX message queue, and the observer design pattern ensures the GUI remains responsive to changes in the detector's state. Screenshots of various tabs in the GUI are provided in figure 2-3 below.

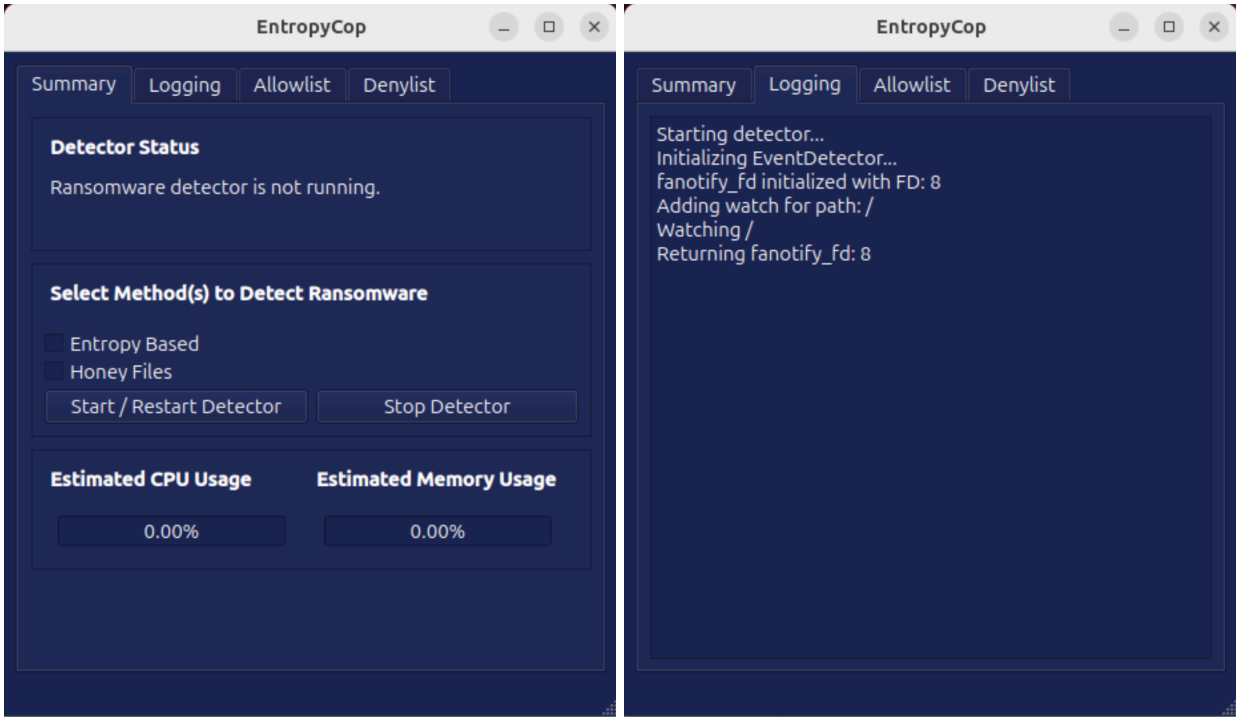


Figure 2. Ransomware Detector GUI's various tabs, from left to right: Summary, Logging

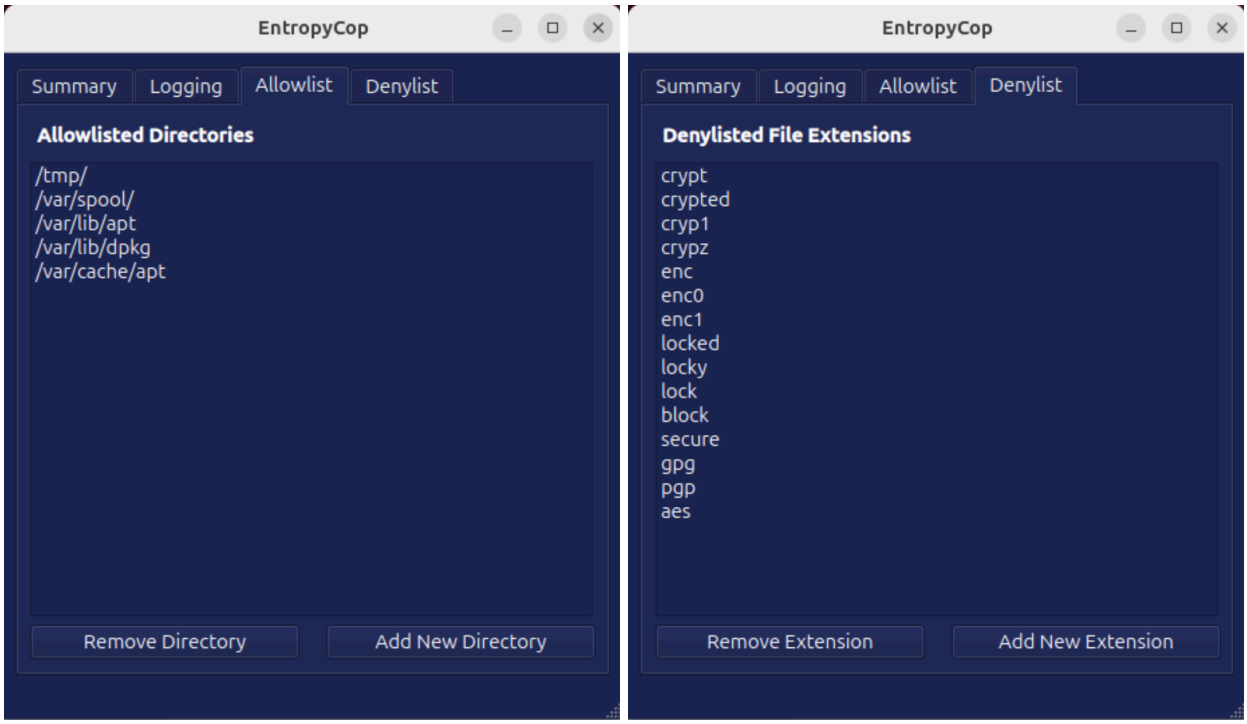


Figure 3. Ransomware Detector GUI's various tabs, from left to right: Allowlist, Denylist

2.4 Admin Monitoring Dashboard

We developed a System Monitoring Dashboard to provide a real-time, event-driven interface for visualizing and managing ransomware activity across a distributed network of client devices. It is built on a WebSocket-based architecture that supports persistent, bidirectional communication between clients and a centralized host server. The backend leverages a SQLite database to track key system metrics, including the number of suspicious processes detected, processes terminated, total connected devices, and the number of compromises (calculated as the difference between suspicious detections and processes killed). Device statuses are updated dynamically and broadcast to all clients upon any state change. The system also maintains a blacklist of SHA256 hashes corresponding to known malicious executables. The React-based frontend presents this information in a responsive layout, displaying the blacklisted hashes alongside aggregate statistics and a grid of connected devices, each labeled with its hostname and current status (e.g., Online, Offline, Threat Detected, Disconnected). When a client detects a threat through the Analyzer module, it notifies the host via Websocket, which then updates the database and propagates the change to all users in real time, ensuring accurate and synchronized system monitoring. When a threat is detected on a device, the dashboard displays a message prompting the host to quarantine the device, however, this functionality is not currently implemented.

The layout of the dashboard is demonstrated in figure 4 below.

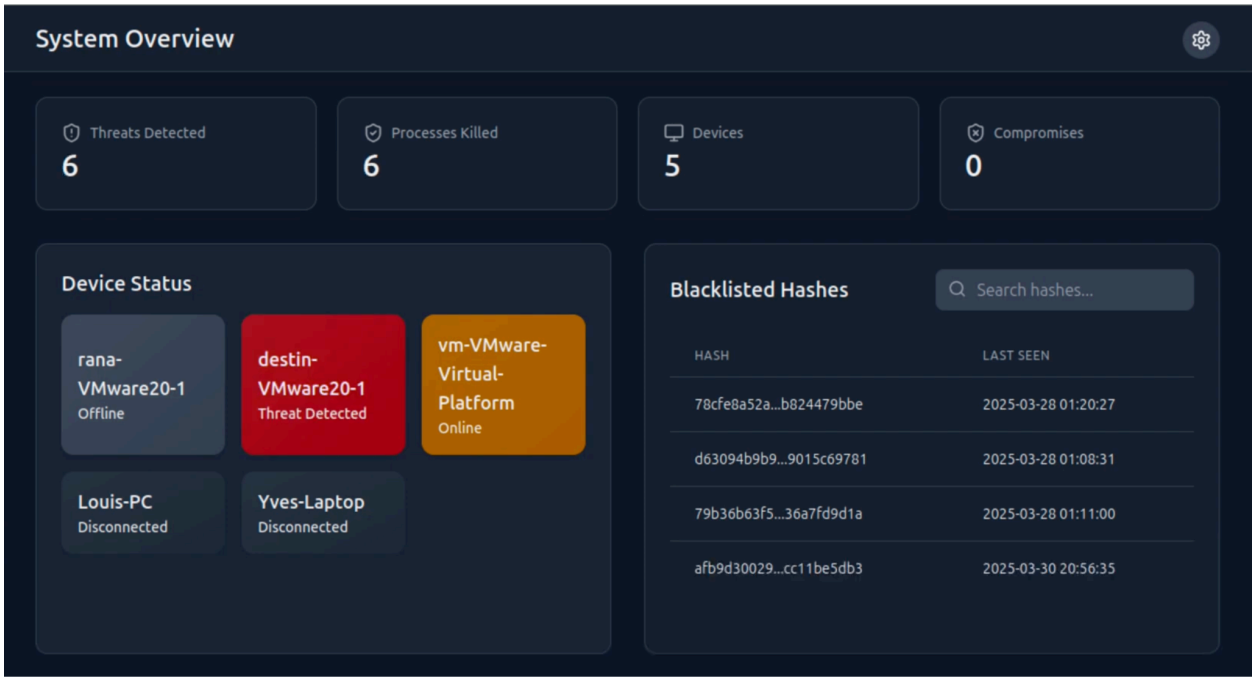


Figure 4. System Monitoring Dashboard

2.5 Problems Encountered

2.5.1 Resolved problems

Overall the development of our project went smoothly, thanks to our unit testing and CI pipeline and adherence to object-oriented best practices from all team members. This ensured that major bugs were rare and always caught quickly. There were still some problems that had to be overcome during design and development, the largest of which are summarized below:

- **Initial class structure:**
  During the early phase of design, we were unsure of the best way to structure the classes of our project in terms of composition and where the “main loop” would be located. This was an important decision as it would be hard to change later and could have made future multithreading significantly more difficult. After conferring with our sponsor and academic advisor, we chose to make the EventDetetor class the owner of the main loop. This decision proved to be correct and made the later change to multithreading easy.

- **False positive rate:**
  Another early problem we noticed is that some non-encrypted files have naturally high entropy. This includes all compressed files, images, video, audio, executables, and some others such as PDFs. These files would be incorrectly flagged by our entropy detector. To solve this issue, we introduced a secondary test called the Monobit Frequency Test for known high entropy file types. Although more expensive to perform compared to entropy calculation, the monobit test is able to robustly distinguish between encryption and

high-entropy regular files. We also had to consider the case of non-suspicious encryptions, such as those done purposely by the user. To this end we implemented an allowlist of directories, which can be edited at any time from the GUI, that will cause the detector to ignore all encryptions made in that directory.

- **Difficulty finding a ransomware sample:**
  Although we developed our own mock ransomware for testing the detector early on, we also wanted to acquire a genuine sample to gauge the effectiveness of our solution in a "real world" scenario. This proved to be difficult, as safety concerns meant that we needed a ransomware whose source code was completely available online so that it could be edited to control certain functionalities. With some fixes, we were eventually able to utilize GonnaCry for our sample ransomware.

- **Encoding handling:**
  As discussed above, a major shortcoming uncovered by testing with GonnaCry was our detector's inability to overcome basic entropy-hiding techniques. To solve this issue we added an extra step to our solution to first check if a file's contents have been base64 encoded, and decoding them before entropy analysis. This was able to defeat GonnaCry's basic entropy hiding, but is not a complete solution (see below).

2.5.2 Unresolved problems

There are some limitations to our chosen detection methods which need to be considered: Our solution relies primarily on entropy analysis, which has proven useful for flagging suspicious encrypted activity but can also be defeated in a number of ways. While the final iteration of our detector includes a check for Base64 encoding, there are many other encoding methods—such as ASCII85, Base32, or URL encoding—that can be used to conceal high-entropy data and evade detection [3]. Even if multiple encoding schemes were checked for, other entropy-hiding techniques exist: "Entropy sharing" attempts to break up encrypted content into smaller, low-entropy chunks that resemble harmless files [4], while "format-preserving encryption" may be able to generate ciphertext that closely mirrors the structure and entropy of regular plaintext, eliminating the need for visible encoding altogether [5]. These techniques collectively make our system susceptible to entropy neutralization attacks. As a result, entropy alone is not a foolproof indicator of ransomware activity, and relying solely on it limits the robustness of our solution. This highlights the need for complementary detection methods to ensure comprehensive protection (see section 3).

2.6 Application Demo

The current working application, along with instructions to run and a demo video, can be found at the following github repository: https://github.com/colecathcart/609Capstone

The following screenshots in figure 5-6 are taken from the application demo video and demonstrate the functionality of the program:

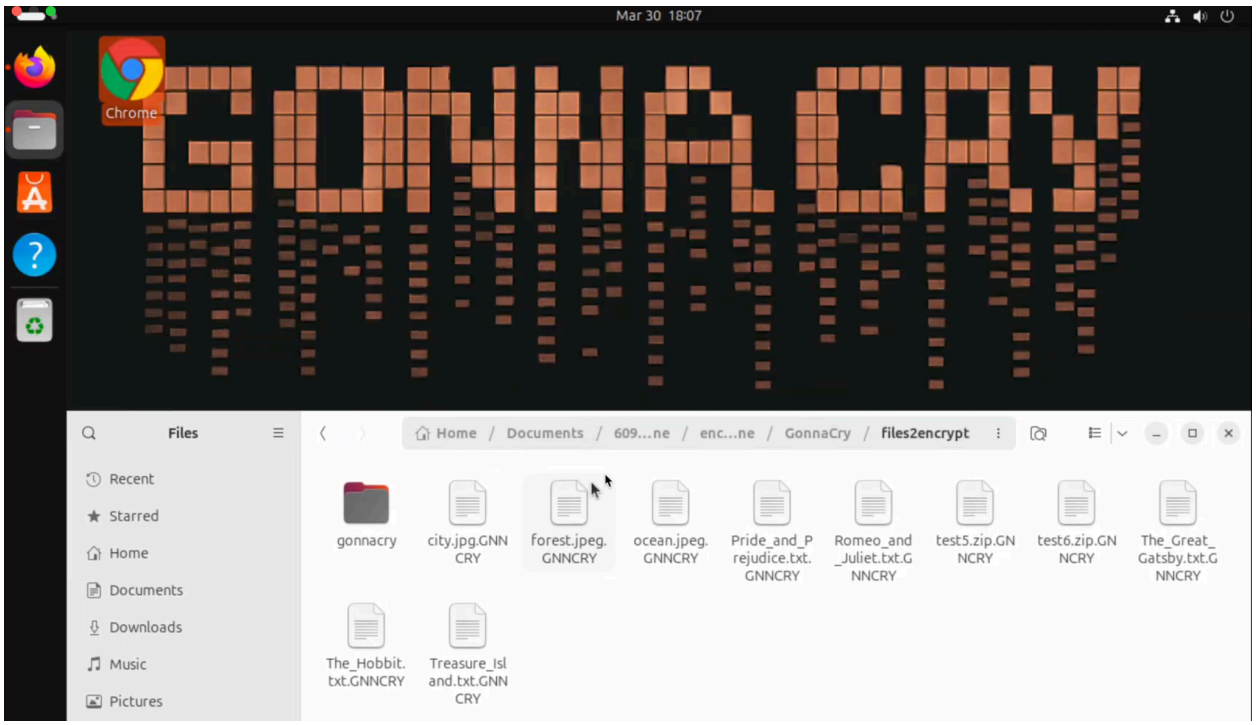


Figure 5. Modified GonnaCry ransomware after encrypting all files in a folder

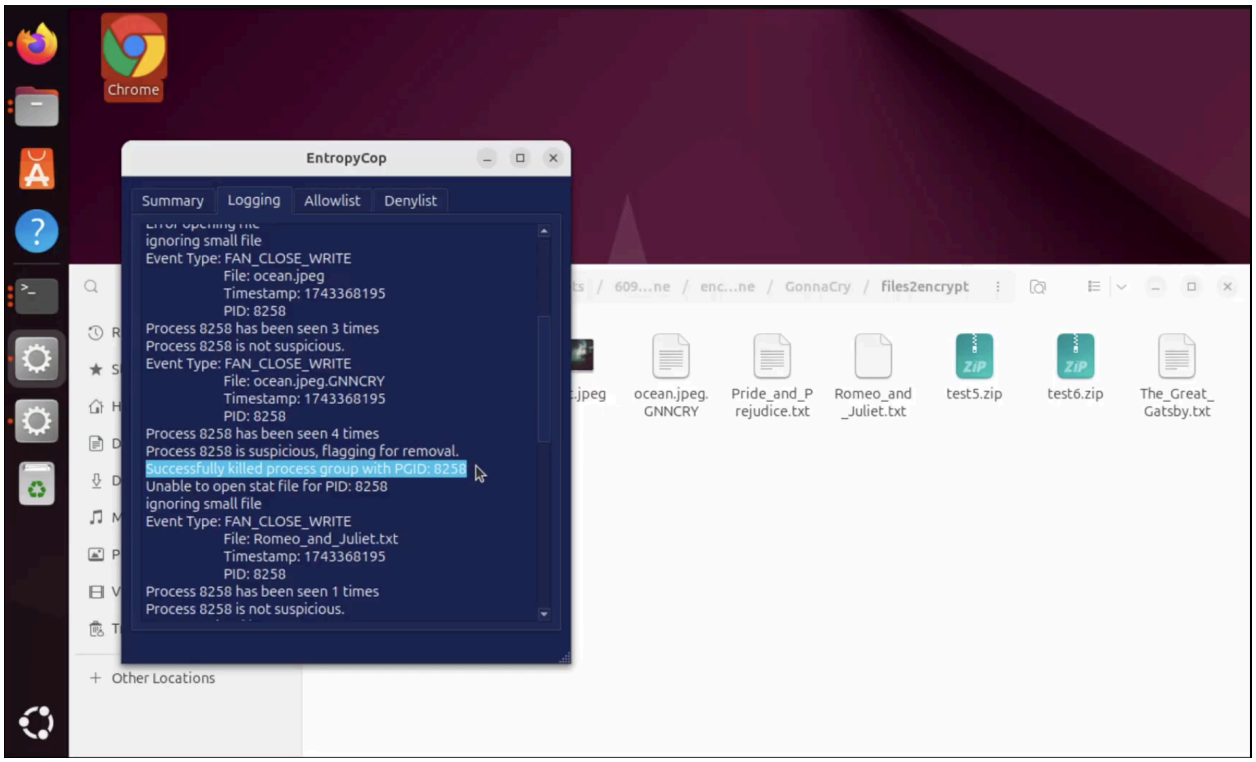


Figure 6. Detector successfully catches and kills the mock ransomware mid-job

2.7 Team Coordination and Workload Sharing

Team progress was managed and tasks were planned primarily through the following methods:

1. Frequent team communication was maintained through a Discord server. A dedicated channel was made for communication with the industry partner.
2. Informal team meetings were held Tuesdays and Thursdays during lecture time, where progress and challenges were discussed.
3. Tasks were tracked through a simple task tracking spreadsheet that included target deadlines and the responsible team member for each task.

In terms of workload sharing, project tasks were primarily completed by members as outlined in table 1 below. While there were members assigned to different tasks, the team ensured to involve others when encountering challenges and to complete code reviews.

Table 1: Project task coordination

Task	Responsible Member
Ransomware Emulator	David
Detector Improvements	Cole
GUI	Destin
Admin Dashboard	Rana

3. CONCLUSIONS AND FUTURE WORK

This project presents a working proof-of-concept ransomware detection and response system tailored for Linux environments. The system leverages entropy-based analysis, real-time file system monitoring via Fanotify and multithreading to efficiently detect and terminate suspicious processes mid-execution. To improve detection accuracy and performance, the Analyzer pipeline incorporates magic byte checking to verify true file types, a tiered entropy sampling strategy for efficiency, and Base64 decoding to mitigate basic entropy-hiding attempts. A Qt-based GUI allows users to start and stop the detector, configure analysis modes, and view output logs in real time. In parallel, a WebSocket-powered admin monitoring dashboard provides a broader view of system activity across devices, displaying threat statistics, device statuses, and blacklisted hashes—all dynamically synchronized through a centralized SQLite database. Together, these components form a cohesive system for real-time ransomware detection and termination.

Despite its success, the system still relies heavily on entropy as the primary detection signal. As discussed earlier, this approach is vulnerable to entropy neutralization techniques such as

alternative encoding schemes, entropy sharing, and format-preserving encryption. While we mitigate some of these with Base64 checks, block-level sampling and magic byte validation, entropy alone is limited in coverage.

For future work, we plan to introduce honey files—decoy files planted throughout the system that trigger alerts when accessed—to help detect ransomware based on behavior rather than file content. We also plan to implement hash-based signature matching, comparing running binaries against a maintained list of known ransomware hashes, which the system already tracks but does not currently act on. The dashboard will also be expanded to allow for quarantining an infected device from the rest of the network. Lastly, we plan to connect our system with existing antivirus tools to enhance response and threat coverage.

4. APPENDICES

Appendix A: References

[1] T. Marinho, *GonnaCry*, GitHub repository, 2018. [Online]. Available: https://github.com/tarcisio-marinho/GonnaCry

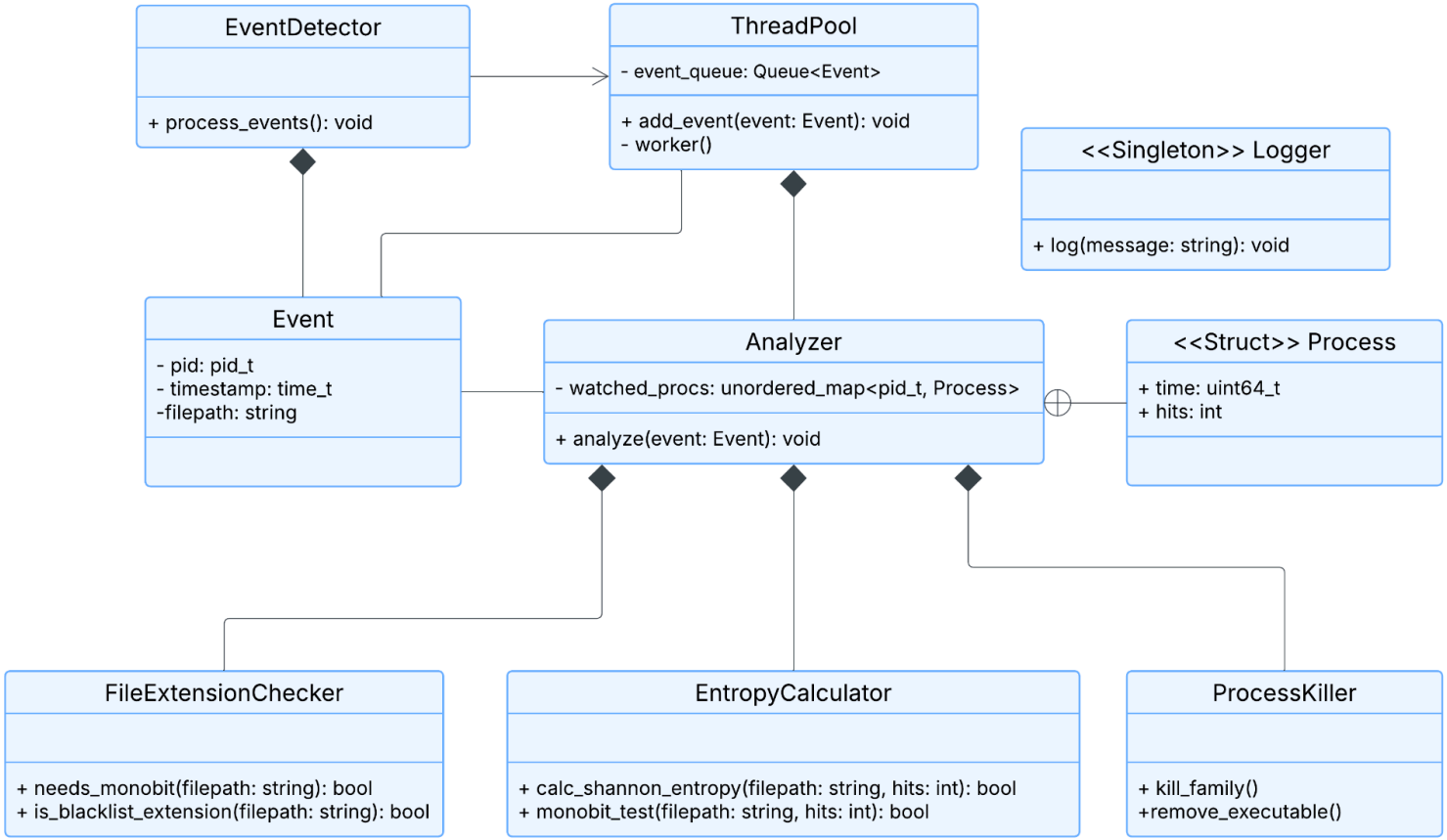
[2] Rukhin *et al*. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *NIST.* Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[3] J. Lee and K. Lee, "A Method for Neutralizing Entropy Measurement-Based Ransomware Detection Technologies Using Encoding Algorithms," *Entropy*, vol. 24, no. 2, p. 239, Feb. 2022. [Online]. Available: https://doi.org/10.3390/e24020239.

[4] J. Bang, J. N. Kim, and S. Lee, "Entropy Sharing in Ransomware: Bypassing Entropy-Based Detection of Cryptographic Operations," *Sensors*, vol. 24, no. 5, p. 1446, Feb. 2024. [Online]. Available: https://doi.org/10.3390/s24051446.

[5] J. Lee, S. Y. Lee, K. Yim, and K. Lee, "Neutralization Method of Ransomware Detection Technology Using Format Preserving Encryption," *Sensors*, vol. 23, no. 10, p. 4728, May 2023. [Online]. Available: https://doi.org/10.3390/s23104728.

Appendix B: UML Diagram



Appendix C: Event State Diagram