

FLORIDA INTERNATIONAL UNIVERSITY

Knight Foundation School of Computing and Information Sciences
Fall 2025 Senior Design Project

Profiling Ransomware Attacks via Web Crawling and Textual Analysis

Students: Jordan Arteaga, Pedro Loor, Luis Salgado, Kevin Montenegro, Elvis Nakamura

Mentor: Weidong Zhu, Assistant Professor

Instructor/Faculty: Seyedmasoud Sadjadi, Florida International University

Project Documentation

This project seeks to analyze the content from ransomware-related news, posts, and reports can provide heuristics for the defense. Profiling the characteristics of emerging ransomware attacks can provide a more effective ransomware defense. This project was built using Cursor as one of its main tools at the suggestions of the project owner to expedite the process as the team and little to no knowledge in python coding or scripting.

Feature list

- Keyword based filtering
- Unfiltered website search
- Self-sufficient web crawling
- Data storage and management
 - Data formatting
 - Pattern classifications

Iterations

Iteration 1:

The first iteration of the project was set to be a self-sufficient website running of Amazon AWS ec2 that could be accessed by anyone on the team and the project manager. This was to be built using python scripting and some pieces of HTML but was quickly scrapped as the project quickly began overreaching into uncovered areas such as gathering malware data samples and running them through Shodan API and VirusTotal but this ultimately missed the mark. Below you will find the original ReadMe for the project's first iteration.

Malware Website Crawler

A comprehensive web application for crawling websites and detecting malware in downloadable files. This system provides automated malware analysis, sandboxing, and reporting capabilities with a modern web interface.

Features

- **Website Crawling:** Automated discovery of downloadable files on websites
- **Malware Analysis:** Static and dynamic analysis of discovered files
- **Sandboxing:** Safe execution environment for suspicious files
- **Database Integration:** PostgreSQL for storing findings and metadata
- **Real-time Monitoring:** Live progress updates during scans
- **Report Generation:** Comprehensive reports with detailed findings
- **AWS Deployment:** Fully containerized and scalable infrastructure

Architecture

The application consists of several key components:

- **Web Application:** Flask-based REST API and web interface
- **Crawler Engine:** Website crawling and file discovery
- **Analysis Engine:** Malware detection and classification
- **Database:** PostgreSQL for persistent storage
- **Cache:** Redis for session management and task queuing
- **Storage:** S3 for file storage and logs
- **Infrastructure:** AWS ECS Fargate for containerized deployment

Iteration 2:

This version scrapped the website design as it simply was too much hassle to get off the ground and took a more simplistic approach in which the crawler and all its necessary functions were kept within Github as it made operations easier. The new problem was that there was too much hardcoded data that made calls to terraform and AWS ECS requesting SQL data that didn't exist as we were no longer going to operate via ECS. The project was then rebuilt, and the old code was scrapped.

Iteration 3:

This version rebuilt the crawler to run via Github workflows and store the necessary data into AWS S3 buckets as the cost was minimal and there was no need for local hosting of SQL or other necessary databases to store the extracted data. There was also Amazon Lambda functionality integration built into the project so that the data could be processed and analyzed all withing AWS. This, however, was another overreach as it began dipping into the tasks given to other team members. The crawler was also built to run with website filtering to allow to known good domains while removing domains that bogged down collection times. This version did not meet expectations and as such was scrapped and picked apart for the last and current version of the crawler. Below you'll find the ReadMes for the third iteration.

Repository Structure & File Tree Guide

This document explains the purpose and organization of each file and directory in the ransomware analysis pipeline repository.

Top-Level Files

Core Documentation

- **README.md** - Main project documentation with quickstart, architecture, and usage instructions
- **SAFETY.md** - Safety guidelines and legal compliance requirements for defensive research only
- **LOGGING.md** - Logging policies and redaction rules to prevent credential/PII leakage
- **README2.md** - This file tree guide (current document)

Configuration & Dependencies

- **pyproject.toml** - Python project configuration with pinned dependencies for reproducibility
- **.gitignore** - Git ignore patterns for Python artifacts, logs, and sensitive files

Configuration Directory (config/)

Runtime-editable configuration files that are synced to S3 during deployment:

- **features.yaml** - Feature extraction rules with keywords and regex patterns for ransomware indicators

- Categories: locking, encryption, exfiltration, ransom_note, initial_access, lateral_movement, C2, ransom_currency, double_extortion, file_extensions_targeted, ransom_family_names

- **suggestions.yaml** - Maps detected features to defensive action recommendations
- **allowlist.txt** - Domain allowlist for crawler (public security sources only)
- **crawler-config.json** - Crawler settings: max_pages, concurrency, delays, user_agent, seed URLs

Core Modules

Storage Layer (storage/)

- **config.py** - AWS configuration dataclass and environment variable loading
- **s3_adapter.py** - S3 storage operations with tagging, lifecycle policies, and raw HTML persistence
- **dynamodb_adapter.py** - DynamoDB operations for document metadata and structured data

Crawler Module (crawler/)

- **init.py** - Module exports
- **settings.py** - Scrapy configuration with robots.txt compliance and rate limiting
- **lambda_handler.py** - AWS Lambda entry point for crawler execution
- **spiders/init.py** - Scrapy spiders module
- **spiders/public_spider.py** - Main crawler spider with allowlist filtering and S3 persistence

Content Extraction (extractor/)

- **init.py** - Module exports
- **extract.py** - Article extraction using Newspaper3k with BeautifulSoup fallback
 - Handles title, text, author, date, canonical URL extraction
 - Language detection and text normalization
- **lambda_handler.py** - Lambda handler for extraction pipeline

Feature Extraction (features/)

- **init.py** - Module exports
- **rules.py** - Rule-based feature extraction from features.yaml configuration
- **lambda_handler.py** - Lambda handler for feature extraction
- **embeddings.py** - Optional ML embeddings using sentence-transformers (flag-driven)

Classification (classifier/)

- **init.py** - Module exports

- **model.py** - Scikit-learn pipeline with TF-IDF vectorization and logistic regression
 - Model training, versioning, and S3 persistence
 - Explainability through feature weights
- **lambda_handler.py** - Lambda handler for classification inference

AWS Infrastructure (aws/)

Deployment & Management

- **deploy.py** - AWS resource creation script
 - Creates S3 buckets with lifecycle policies
 - Sets up DynamoDB tables with on-demand billing
 - Creates AWS Budgets with SNS alerts
 - Tags all resources for tracking
- **teardown.py** - Complete resource cleanup script
 - Deletes all tagged resources (S3, DynamoDB, etc.)
 - Ensures no orphaned resources remain

GitHub Actions (.github/workflows/)

All workflows are manual-only (workflow_dispatch) for safety:

- **deploy.yml** - Manual deployment workflow
 - Dry-run cost estimation
 - Resource creation with budget controls
 - Requires explicit confirmation
- **destroy.yml** - Manual teardown workflow
 - Requires "I want to destroy" confirmation
 - Runs aws/teardown.py
- **run-crawl.yml** - Manual crawl execution
 - Configurable page limits and run modes (lambda/EC2)

Documentation (DOCS/)

- **architecture.md** - High-level system architecture and component interactions
- **deployment.md** - IAM permissions, workflow usage, and security considerations
- **features.md** - Feature configuration editing and runtime behavior

- **testing.md** - Test organization and execution instructions

Optional Web UI (webui/)

- **index.html** - Minimal static viewer for analysis results
 - Read-only interface for browsing classifications
 - Displays confidence scores, features, and suggestions
 - Designed for S3 static hosting with optional CloudFront

Testing (tests/)

Unit tests for all major components:

- **test_features.py** - Feature extraction rule testing
- **test_classifier.py** - Model training and prediction testing
- **test_crawler_allowlist.py** - Crawler allowlist filtering
- **test_deploy_dryrun.py** - Deployment cost estimation testing

Examples (examples/)

Sample data and expected outputs:

- **sample1.txt** - Example article text for testing
- **expected1.json** - Expected JSON output format showing:
 - URL, title, date, extracted text
 - Feature detection results
 - Classification with confidence
 - Defensive suggestions
 - Model version and S3 paths

Module Dependencies

Core Dependencies (requirements.txt)

- **boto3** - AWS SDK for Python
- **scrapy** - Web crawling framework
- **beautifulsoup4** - HTML parsing
- **newspaper3k** - Article extraction
- **scikit-learn** - Machine learning pipeline
- **pyyaml** - Configuration file parsing

Optional Dependencies

- **sentence-transformers** - ML embeddings (flag-driven)
- **spacy** - NLP processing (flag-driven)
- **pytest** - Testing framework
- **moto** - AWS service mocking for tests

Data Flow Architecture

Seeds/Allowlist → Crawler → S3 Raw → Extractor → DynamoDB → Features → S3 → Classifier → DynamoDB/S3 → Web UI

1. **Crawler** reads allowlist and seeds, respects robots.txt, stores raw HTML in S3
2. **Extractor** processes HTML, extracts article content, stores metadata in DynamoDB
3. **Features** applies rule-based extraction using features.yaml, stores results in S3
4. **Classifier** runs ML inference, stores predictions with explainability in DynamoDB/S3
5. **Web UI** provides read-only access to results

Security & Compliance

- All AWS resources are tagged for tracking and cleanup
- Budget controls prevent cost overruns (\$5 default cap)
- Logging redacts secrets and PII
- Only public, non-sensitive sources are crawled
- No malware samples or private data collection

Cost Controls

- S3 lifecycle policies (30-day archival/deletion)
- DynamoDB on-demand billing
- Small Lambda sizes (128-512 MB)
- Optional EC2 (t3.micro) with explicit confirmation
- AWS Budgets with SNS alerts

Iteration 4:

The final iteration of the project utilizes keyword based searches and removes the need for website filtering as this was a personal request of the project owner as it ensured any and all data that could be of use is collected and anything that was deemed useless could be removed during data processing. The data was then processed and collected into AWS and a Github workflow was created to turn the raw data into three file formats: JSON, at the request of the project owner, CSV, a personal addition to make data collection and combination easier as it would ensure data duplication was removed if missed, and Markdown, which was an inclusion by the LLM agent as it was stated to be best suited for LLM analysis. Below you will find the final ReadMe for this iteration.

Ransomware Analysis Pipeline (Simplified, S3-Only)

IMPORTANT: Defensive research only. This project only crawls and analyzes public, non-sensitive sources (news, vendor blogs, public incident writeups). It must never be used to retrieve malware binaries, private/internal documents, credentials, or to perform intrusive scanning. See SAFETY.md.

Overview This repository provides a simplified, defensive ransomware-analysis pipeline that runs continuously and stores data in S3. It uses keyword-based filtering instead of allowlists and can be terminated via GitHub workflows.

Key Capabilities

- Simplified deployment using only S3 storage (no DynamoDB)
- Continuous crawler with keyword-based filtering (no allowlist required)
- Duplicate detection prevents re-crawling of existing websites
- Data stored with website names in month/day folder structure
- GitHub workflow for graceful crawler shutdown
- Daily data combination for LLM processing
- Feature extraction from editable features.yaml; rule-based keywords/regex
- Model and configuration artifacts stored/versioned in S3; runtime-editable configs (features.yaml, suggestions.yaml, crawler-config.json)
- Budget guard: default \$5 monthly AWS Budget; all resources tagged and deletable

Safety & Compliance

- Public information only; respect robots.txt and site TOS
- No malware samples, credential harvesting, or intrusive activities
- Logging redacts secrets and PII; avoid printing full raw articles in CI logs

Architecture (Simplified) ASCII Diagram

[Keyword Discovery] --> [Continuous Crawler] --> [Keyword Filtering] --> [Duplicate Detection] --> [S3 Processed Bucket] | | | v v v [Feature Extraction] --> [Daily Data Combiner] --> [LLM-Ready JSON Files] | | v v [Relevance Scoring] --> [LLM Analysis Pipeline]

The crawler runs continuously with duplicate detection and can be terminated via GitHub workflows.

Cost Controls

- Default budget cap: \$5/month via AWS Budgets API (configurable in deploy workflow)
- S3-only storage with lifecycle policy (archive or delete after 30 days by default)
- No DynamoDB costs

Repository Layout

- features.yaml, suggestions.yaml: editable, synced to S3
- crawler/: crawler workers and configuration adapters
- extractor/: article parsing and normalization
- features/: rule-based feature extraction
- classifier/: training and inference with explainability
- storage/: S3 adapter only
- aws/: deployment, teardown, shutdown, and data combination scripts
- .github/workflows/: deploy.yaml, destroy.yaml, shutdown-crawler.yaml, combine-daily-data.yaml
- tests/: pytest unit tests and dry-run AWS tests
- examples/: sample excerpts and expected outputs
- DOCS/: deeper documentation for modules and operations