

**PROVENANCE GRAPH GENERATION FOR INTRUSION DETECTION**

**PERI ADHITYAN**

**SUUPERVISOR: A/P KE YIPING KELLY**

**COLLEGE OF COMPUTING AND DATA SCIENCE**

Submitted in Partial Fulfilment of the Requirements for the Bachelor of Computing (Computer Science) Nanyang Technological University

**2025**

Table of Contents

[Abstract 4](#_Toc187490077)

[Acknowledgements 5](#_Toc187490078)

[List of Figures 6](#_Toc187490079)

[List of Tables 7](#_Toc187490080)

[Introduction 8](#_Toc187490081)

[Background 8](#_Toc187490082)

[Objectives 9](#_Toc187490083)

[Literature Review 10](#_Toc187490084)

[Data Provenance 10](#_Toc187490085)

[Intrusion Detection Systems 10](#_Toc187490086)

[Provenance Based Intrusion Detection Systems 11](#_Toc187490087)

[KAIROS 11](#_Toc187490088)

[FLASH 12](#_Toc187490089)

[UNICORN 12](#_Toc187490090)

[Provenance Capture Systems 12](#_Toc187490091)

[CamFlow Features 12](#_Toc187490092)

[CamFlow Provenance Capture System 13](#_Toc187490093)

[Setting Up CamFlow 13](#_Toc187490094)

[CamTool Installation 16](#_Toc187490095)

[Provenance Graph Visualisation 16](#_Toc187490096)

[Flurry Framework 17](#_Toc187490097)

[Background 17](#_Toc187490098)

[Flurry Installation 17](#_Toc187490099)

[Cloning Flurry Source Code 17](#_Toc187490100)

[CamFlow Reconfiguration 18](#_Toc187490101)

[XAMPP 18](#_Toc187490102)

[DVWA 19](#_Toc187490103)

[MQTT 21](#_Toc187490104)

[Additional Dependancies 23](#_Toc187490105)

[Running Flurry 24](#_Toc187490106)

[Security-Critical Applications 26](#_Toc187490107)

[Intrusion Scenarios 27](#_Toc187490108)

[XSS 27](#_Toc187490109)

[Stored 27](#_Toc187490110)

[Reflected 27](#_Toc187490111)

[DOM 27](#_Toc187490112)

[Command Injection 27](#_Toc187490113)

[SQL Injection 28](#_Toc187490114)

[Brute Force 28](#_Toc187490115)

[Provenance Graph Generation 28](#_Toc187490116)

[Choosing Dataset Sizes 28](#_Toc187490117)

[Generating Provenance Data 28](#_Toc187490118)

[Graph Exploration 28](#_Toc187490119)

[Benign and Malicious Graph Comparison 30](#_Toc187490120)

# Abstract

Cyber intrusion has been a growing issue for anyone with a digital footprint from individuals to companies to countries. This gives rise to the dire need for urgent and robust intrusion detection systems to pre-empt and mitigate cyber security incidents before major damage can be done. Data provenance graphs are being researched and utilized in auditing and intrusion detection for cyber security. Provenance graphs can depict the entirety of system execution and assist in gathering information regarding the origin of data, the current state and the entities that acted upon it. This project aims to setup existing provenance capture systems to generate and capture provenance graphs during benign system execution and simulated attack scenarios. Additionally, the generated graphs will be used to train and test models to develop intrusion detection systems that can be studied for real-world application.

# Acknowledgements

I would like to express my gratitude for my final year project supervisor, Associate Professor Ke Yi Ping, Kelly for her mentorship, guidance and assistance throughout this project.

Additionally, I would like to thank PHD Student Mr. Li Wei for taking the time to answer queries and offer advice with regards to this project.

# List of Figures

# List of Tables

# Introduction

## Background

The world we live in today is one that relies on the virtual cyber space as much as it does on the physical space, some might say even more so. But the exponential growth in cyber activity has led to the direct increase in cyber-attacks as well. An increased digital footprint gives way to an array of attack vectors for threat actors to use as an entry point. Year by year we see an increase in cyber-attacks, with 2023 showing a 72% increase in data breaches surpassing the previous record high in 2021 [[Forbes]](https://www.forbes.com/advisor/education/it-and-tech/cybersecurity-statistics/).

To counteract these breaches and detect intrusion, researchers have turned to the usage of a provenance graph-based system. Provenance graphs are directed acyclic graphs used to determine relations between entities such as sockets, files and users, and actions such as the flow of data between them. Constructed from system-level logs, these graphs describe the interactions between kernel objects to represent system execution history in a structured manner [[paper]](https://arxiv.org/pdf/2308.05034). These give an insight as to what benign activity might look like as compared to when a threat actor has breached the system for malicious intent. Researchers from Harvard and Cambridge Universities have pointed out the extensive capture of security sensitive kernel operations, the explicit relations it depicts between objects, that intrusions result from unexpected relations and the robustness of graphical representation [[Paper]](https://www.seltzer.com/assets/publications/Provenance-Based-Intrusion-Detection.pdf).

Provenance graphs show great advantages when it comes to intrusion detection. Since provenance graphs show system execution by displaying relations between system objects, simple audit files that are unstructured and hard to read can be converted to provenance graphs. Secondly, provenance graphs are hard for attackers to replicate or forge as they are rich in semantics. They take into consideration spatial and temporal information which allow security analysts to conduct thorough and effective investigations. Finally, provenance graphs store all the execution history which aid analysts in investigating Advanced Persistent Threats (APTs). APTs are known for their long term embedding in systems and stealth in being undetected. The complete history of system execution provided by provenance graphs can easily aid analysts in the event of APTs [[Paper]](https://www.sciencedirect.com/science/article/abs/pii/S0167404821001061).

However, these discoveries and techniques do not come with their downsides. Recent Provenance Graph Based Intrusion Detection Systems (PIDS) used embedding techniques that incur high computational resource cost. Furthermore, as these systems take inputs from graphs, there are detection delays. Finally, these systems output uninterpretable results that do not give much detail other than the fact that they have been flagged out because they deviate from normal system operation. [[Paper]](https://arxiv.org/pdf/2404.14720).

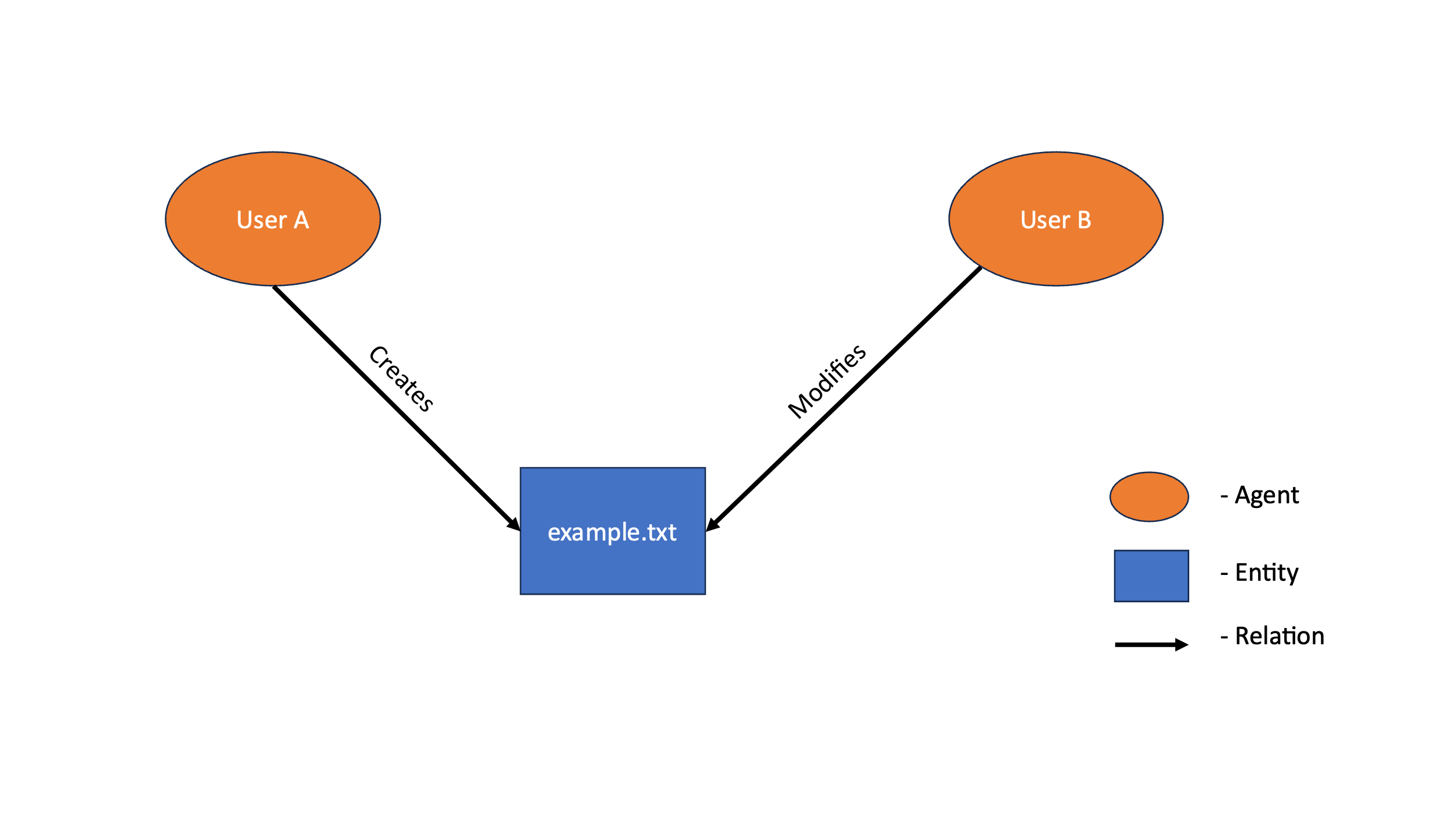
## Objectives

This study aims to setup a provenance capture system to capture whole system provenance. It will leverage CamFLow, an open-source project to bring observed provenance collection to the Linux Operating System that complies with W3C PROV-DM standard. These provenance graphs will be generated by implementing the FLURRY framework, an end-to-end data pipeline which simulates cyberattacks captures provenance data from these attacks into data provenance graphs and incorporates this data with a framework for training deep neural models that supports preconfigured or custom-designed models for analysis in real-world resilient systems[[paper]](https://arxiv.org/pdf/2203.02744) Finally, it will utilize these generated graphs to train, test and gather statistics from models for benchmarking and further improvement.

# Literature Review

## Data Provenance

Data provenance in the context of computer systems is the description of relationships between entities, the activities conducted using those entities and the agents associated with them. For example, if *user A* creates a file *example.txt*, followed by *user B* editing the file, *example.txt*, data provenance will give us these relations from the creation, all the way through till the file is destroyed. *Figure 1* depicts how a simple data provenance graph can be generated based on the earlier scenario.



**Figure 1: Simple provenance graph showing relations between agents and entities**

## Intrusion Detection Systems

Known Intrusion Detection Systems (IDS) scrutinise different data sources and can be deployed at the network or host level [[paper]](https://dl.acm.org/doi/pdf/10.1145/3539605). Albeit there existing a plethora of IDS available for use, current IDS succumb to a large number of false positives when it comes to detecting intrusions [[paper](https://dl.acm.org/doi/pdf/10.1145/3539605), [paper](https://www.seltzer.com/assets/publications/Provenance-Based-Intrusion-Detection.pdf)]. These false alarms may overwhelm security analysts and divert their attention toward benign actions while malicious actions by threat actors go undetected or buried within these alarms.

Network IDS (NIDS) monitor network traffic of the various hosts on the network layer and are easily merge into existing network infrastructure. However, NIDS cannot analyse encrypted network traffic, common mode of communication for threat actors to communicate with Command and Control (C2) servers or data exfiltration. Additionally, NIDS can only detect external intrusions, while internal compromises might not be flagged [[paper]](https://dl.acm.org/doi/pdf/10.1145/3539605).

A step up from this would be the usage of Host based IDS (HIDS). HIDS identify intrusions by monitoring file systems in addition to network events. The additional or more fine-grained information gathered from event logs allow HIDS to outperform NIDS. However, HIDS would need to be deployed on every host, can only monitor the particular host that it was deployed on and generate large amounts of logs that require storage [[paper]](https://dl.acm.org/doi/pdf/10.1145/3539605).

Signature based detection systems are crafted based on security experts’ knowledge on malicious behavior from past attacks. These work well for known cyber-attacks but fall short when a new, undiscovered attack surfaces. The performance of signature-based systems solely relies on the diversity of the dataset it references[[paper]](https://dl.acm.org/doi/pdf/10.1145/3539605).

This gives way to the rise in popularity for Provenance based IDS (PIDS). State-of-the-art (SOTA) whole-system provenance capture systems leverage the Linux Security Module (LSM) interface to record provenance for every security related interaction. Provenance data depict system execution as interactions between data objects. Using these features made available from data provenance, intrusions can be detected by finding deviations from normal interactions [[paper]](https://www.seltzer.com/assets/publications/Provenance-Based-Intrusion-Detection.pdf).

Usage of provenance data allow for detection of intrusion due to the propagation of events. Even though an attacker uses a single point of entry, the actions conducted by them propagate through the system establishing more connections to other objects resulting in anomalous behaviour. For example if the attacker executes a script that collects documents, to exfiltrate said data to an external server, many relations between entities are created which deviate from benign behaviour indicating malicious intent.

Furthermore, the robust graph representation is much more complex to replicate to depict benign behaviour by the attacker, making it difficult for them to camouflage their actions.

## Provenance Based Intrusion Detection Systems

Recent developments have led to multiple SOTA PIDS available for testing and benchmarking. This section will be highlighting a few with their features and advantages.

### KAIROS

KAIROS leverages all kennel interaction of an entire network of systems, as opposed to application specific data. The information gathered from the whole-system allows KAIROS in detecting modern sophisticated intrusions such as APTs as they affect multiple applications on a host before migrating to others on the network. Like other PIDS, KAIROS considers attackers maintaining persistence within a system by exploitation and placing backdoors for communication. As an anomaly-based intrusion detection system, KAIROS learns from provenance graphs of benign system execution. It has been acknowledged that if system behaviour changes, KAIROS may not be as affective in detecting anomalies or might report false positives [[paper]](https://arxiv.org/pdf/2308.05034).

### FLASH

FLASH, harnesses contextual and structural information from data provenance graphs to improve detection rates. It tackles the issue of overlooking semantica information during detection by utilising a Word2Vec-based embedding technique. This technique encodes various node attributes found in provenance graphs and transforms them into semantically rich feature vectors. Additionally, it is able to obtain temporally sensitive embeddings and therefore including temporal ordering among events [[paper]](https://dartlab.org/assets/pdf/flash.pdf).

### UNICORN

Mainly focussed on countering APTs, UNICORN uses graph sketching to build and updatable and incrementable graph data structure that enables efficient computation of graph statistics. The longitudinal nature of the graph structure allows UNICORN to track long term stealthy intrusions. To make it scalable and require less computational and storage resources, UNICORN implements a fixed size and incrementally updatable graph structure [[paper]](https://arxiv.org/pdf/2001.01525).

## Provenance Capture Systems

Over the years, researchers have developed various provenance data capturing systems and libraries. This study will utilize CamFlow, an open-source project to bring observed provenance collection to the Linux Operating System. The upper hand CamFlow has compared to other provenance capture systems, is that CamFlow uses a self-contained, easily maintainable implementation relying on a Linux Security Module, NetFilter and other existing kernel facilities. This provides a mechanism to tailor the captured provenance data to the needs of the application, making it easy to integrate provenance across distributed systems [[paper]](https://arxiv.org/pdf/1711.05296).

## CamFlow Features

CamFlow captures a system’s provenance data through *Linux Security Model* and *NetFilter* hooks which is subsequently transferred to user space for storage and analysis. CamFlow tackles the major concern of generating extremely large whole-system provenance by providing mechanisms that capture only necessary subsets of the entire provenance graph required by applications. It further allows the possibility to limit the capture to specific edge or node types and flows from specific sources such as inodes, network interfaces, security contexts and user IDs [[paper]](https://camflow.org/posters/socc-2017.pdf).

# CamFlow Provenance Capture System

## Setting Up CamFlow

The first task for this project would be to setup CamFlow on a system in order to capture whole system provenance. CamFlow is recommended to be installed on Fedora Linux which is based on the Linux OS kernel architecture. For maximal compatibility, a virtual machine running Fedora Linux was setup with the following configurations:

Host OS: Windows 11

Hypervisor: VirtualBox 7.1

Virtual Machine OS: Fedora 35 64-bit

Memory: 8GB

Storage 100GB

Referencing the CamFlow project website [[site]](https://camflow.org/), the fastest installation method was used, downloading the package manager and installing the camflow package using the following commands



After which to activate the services, the following commands need to be run



Finally, reboot the system. Hitting ‘shift’ when the VirtualBox splash screen appears will bring the system to the boot menu, where the kernel option with the word ‘camflow’ should be chosen. Rebooting the system should set camflow as the default kernel to boot from, however in the event that does not occur the following command can be run within a terminal:



This will list the existing kernels installed, simply find the kernel with ‘camflow’ in it’s title:



**Figure 2: Screenshot of the results after listing existing kernels**

After identifying the correct kernel simply run the following command to ensure that the default kernel is set:



Ensure that the kernel is the correct one according to the system being setup. In the context of this project, installation scripts were created to facilitate multiple VMs being setup with ease. The entirety of the script will be listed in the appendix below



## CamTool Installation

CamTool is an extension of CamFlow that helps with the visualization of simple provenance graphs. The Message Queuing Telemetry Transport (MQTT) protocol is used to publish provenance in real time on CamFlow’s demo website (<https://camflow.org/demo>). Another script was used for the installation of CamTool as shown below. To collect logs in the w3c format the *camflowd.ini* file needs to be edited, followed by restarting the *camflowd.service* service.



## Provenance Graph Visualisation

To visualize provenance using CamFlow and CamTool, a simple scenario of file creation and deletion was used. In a browser, open the demo website and click ‘Start CamFlow MQTT’, ensure the browser is open throughout the demonstration. To capture and track the provenance of a file from creation to deletion the following commands were issued:



On the demo website the following provenance graph can be visualized tracing the example.txt file from creation to deletion.



**Figure 3: Provenance graph generated from example.txt**

# Flurry Framework

## Background

To bridge the gap between generating usable provenance data and utilizing these graphs for unsupervised learning to detect intrusions, researchers have developed Flurry. Flurry is equipped with provenance capture, graph generation tools and can execute automated cyber-attacks and collect provenance graphs that can be used as input for machine learning tools. [[Paper]](https://arxiv.org/pdf/2203.02744) The benefits provided by this framework are that it is an end-to-end pipeline that takes in system execution and models it as a multi-layer provenance graph, displays execution of cyber-attacks, and provides a plug-and-play framework for graph learning models to analyse provenance graphs.

## Flurry Installation

The author of the Flurry Framework, Maya Kapoor, has provided scripts that automated the download and installation of the needed dependencies through her git repository <https://github.com/mayakapoor/flurry> . However, it has since been deprecated and requires manual installation of the following dependencies to run as intended, listed below:

1. XAMPP
2. DVWA
3. MQTT
4. Additional Dependencies

Scripts have been written to ease the installation of Flurry and its dependencies.

### Cloning Flurry Source Code

The first steps are to clone the source code from the git repository followed by resetting it to the last working version of the framework



Followed by cloning *flake* into the created *flurry* folder and configuring the code as shown in the script in the appendix.



### CamFlow Reconfiguration

To ensure compatibility between CamFlow and Flurry, the following changes need to be made to configuration files:

*camflowd.ini*

1. output=mqtt
2. format=w3c
3. address=localhost:1883
4. password=camflow
5. qos=0

*camflow.ini*

1. ;all=false
2. Duplicate=true

The automated script displaying the entire configuration files and changes is listed in the appendix for clarity.

### XAMPP

XAMPP is a popular PHP development environment containing, MariaDB, PHP and Perl. It is an open-source package that helps a local host test websites amongst other use cases[[site](https://www.apachefriends.org/), [site](https://www.javatpoint.com/xampp)].

Installing the necessary dependencies:



Download the XAMPP executable from <https://www.apachefriends.org/> and installing it:



Finally complete the installation with the GUI.

### DVWA

Damn Vulnerable Web Application (DVWA) is a deliberately insecure web application that allows testers to penetrate known vulnerabilities of web applications[[site](https://owasp.org/www-project-damn-vulnerable-web-sockets/), [github](https://github.com/digininja/DVWA)]. In the scope of this project, the exploitations of various vulnerabilities and their benign counter scenarios are automated. These can be used to generate provenance data for benign and malicious scenarios. Due to its vulnerability, it is intended to be installed locally as opposed to a public web server.

The following script was used to clone and configure DVWA to ensure its compatibility with the Flurry framework.



After installation, navigate to <http://localhost/dvwa/setup.php> and click on ‘Create/Reset Database’

A screenshot of a computer

Description automatically generated

**Figure 4: Screenshot of the DVWA home page**

Login with the following credentials

Username: admin

Password: Password

A screenshot of a login screen

Description automatically generated

**Figure 5: Screenshot of the DVWA login page**

The setup page should be displayed once successfully authenticated

A screenshot of a web application

Description automatically generated

**Figure 6: Screenshot of the DVWA setup page**

### MQTT

Message Queuing Telemetry Transport is a messaging protocol designed for efficient, relaible communication between systems over local networks. Flurry utilises Mosquitto MQTT to stream provenance data [[site]](https://mosquitto.org/). It utilises the publish and subscribe architexture that eases the distribution of messages. Flurry subscribes to CamFlow via a designated topic and as such MQTT needs to be run inorder for Flurry to be functional. The following script was used for the installation:



The following commands can be used to run and kill the MQTT service respectively.



### 

### Additional Dependancies

The final portion for the installation of Flurry will be to initialise a virtual environment and install the necessary dependencies. Google Chrome and the latest Chrome driver must be installed for the automated running of DVWA. A virtual envrionment with python3.6 is initialised in order to ensure compatibility with modules devoloped using an older python version. The following link was used to find the latest stable version of chrom driver to be downloaded and installed. The following script was used for the virtual environment creation, Chrome driver installation and dependancy installation necessary.



### Running Flurry

Prior to running Flurry, XAMPP and MQTT need to be launched which can be done by running these commands

****

Flurry’s authors have written a script, *webserver.py*, to execute benign and malicious scenarios. This script conducts various automated actions, captures the provenances data and outputs it into text files, JSON graph files and a png file in order to visualise the graph generated. For this demonstration, a simulation of a message board post and the capture of whole system provenance was selected. Additionally, fine granularity for edge types and node types were selected. The figures below show the command line interface and output of the Flurry framework after a single iteration.

A screenshot of a computer screen

Description automatically generated

**Figure 7: Screenshot of the Flurry command line interface**

A screenshot of a computer

Description automatically generated

**Figure 7: Screenshot of the output of Flurry files**

# Security-Critical Applications

Application security can be broken down into three main branches. Web, API and Cloud. This project will be generating whole system provenance generated by DVWA and hence, will be focusing on the criticality of secure web applications.

Web applications are software that tun on web servers and are accessible through the internet. These applications naturally accept connections from clients over insecure networks which expose them to various vulnerabilities. Web applications succumb to some of the most severe cyber-attacks as they are business critical and contain sensitive data. The Open Web Application Security Project (OWASP) have published the OWASP Top 10 which is a collection of the most severe and common vulnerabilities that web applications are prone to [[paper](https://www.imperva.com/learn/application-security/application-security/)]. These include:

|  |  |
| --- | --- |
| OWASP Top 10 | Description |
| Broken access control | Enables attackers to gain unauthorised access to user accounts or provides users with unauthorised privileged functions |
| Cryptographic Failures | Occurs when data is not effectively secured while in transit and storage. This can lead to the unwanted distribution of passwords, health records and other such personal and private information. |
| Injections | These vulnerabilities allow attackers to send malicious data to web application interpreter which can be compiled and executed on the server. There are various forms of Injection based attacks such as Cross Site Scripting, SQL Injection, Cross Site Request Forgery and more. |

# Intrusion Scenarios

The DVWA provides a platform for users to test their penetration skills using attacks such Cross Site Scripting (XSS) and their various forms, reflected, DOM and stored, brute force attacks, command injection and so on. Flurry can automate these attacks and their benign counterpart to capture provenance data for both scenarios. Below lists each malicious scenario and their corresponding benign action.

## XSS

Cross Site Scripting or commonly known as XSS is a vulnerability found in web security that allows attackers to compromise interactions between users and a web application. They work by injecting malicious scripts into trusted websites which allow the attacker to access data such as cookies, session tokens or other sensitive information.

### Stored

These attacks are when the injected script is permanently stored on target servers such as in databases or message forums and are retrieved by victims when they request the stored information[[site](https://owasp.org/www-community/attacks/xss/)]. Flurry simulates this and its benign version in the form of a message board post.

### Reflected

XSS Reflected attacks occur when applications receive data in an HTTP request which is included in the response immediately in an unsafe manner [[site](https://portswigger.net/web-security/cross-site-scripting/reflected)]. Reflected attacks can be delivered to victims in the form of emails tricking them into clicking malicious link or merely browsing a malicious site. The injected code travels to a vulnerable compromised by the attacker which reflects back to the victim’s browser [[site](https://owasp.org/www-community/attacks/xss/)]. The benign version simulated in this project will be completing a questionnaire.

### DOM

Document Object Model (DOM) is an internal data structure that stores objects and properties of a webpage which is taken advantage of during an XSS DOM based attack [[site](https://www.invicti.com/learn/dom-based-cross-site-scripting-dom-xss/)]. The attack modifies the DOM “environment” in a victim’s browser to run the client-side script in an unexpected manner. Flurry’s benign simulate of an XSS DOM based attack is querying a webpage.

## Command Injection

Command Injection attacks’ goal is the execution of arbitrary commands on the operating system of the host through the vulnerable web application. It exploits unsafe user applied data such as forms and cookies to a system shell and executes system commands with the privileges of the exploited application [[site](https://owasp.org/www-community/attacks/Command_Injection)]. Applications may be exposed if they lack sufficient input validation. The benign version of this attack to be simulated will be the pinging of the local host.

## SQL Injection

SQL injection allows an attacker to access sensitive data from databases through carefully constructed SQL commands that reveal user information. Some injections may be used to modify or delete data, in addition to performing denial-of-service attacks [[site](https://portswigger.net/web-security/sql-injection)]. The benign version of this attack would be a regular database entry.

## Brute Force

Brute Force attacks are simply the attacker guessing user credentials by exhausting all possibilities of combinations or a curated attack dictionary [[site](https://www.fortinet.com/resources/cyberglossary/brute-force-attack)]. The benign version of this attack would be a simple login.

# Provenance Graph Generation

## Choosing Dataset Sizes

The flurry framework provides a script that automates each scenario individually or one after another with the possibility for setting the number of iterations per run. However the scope of this project is to generate different data to simulate benign system execution and malicious, which meant unique orders of the scenarios. To do that, scripts were written to permute scenarios. In total, there were 1956 unique benign permutations of all six benign scenarios. In addition to the unique benign permutations, repeated runs were added to make it a total of 2000. Next, to permute a malicious scenario with different sets of benign scenarios would results in running 70,000 runs of Flurry which would not have been feasible. Hence, 2000 random combinations for each malicious scenario were chosen. The scripts used to permute the scenarios and split them are in the appendix.

## Generating Provenance Data

To generate the provenance data, scripts were written to automate Flurry to run different permutations of benign and malicious actions. To prepare the system for easy generation of data, the requirement for entering the root password was disabled. Multiple clones of the virtual machine with the same configurations were set up to concurrently collect the provenance graphs to streamline the process and collect the data faster.

## Graph Exploration

Flurry outputs multiple files once provenance data is collected for each run. These include json files for node types, edge types, the graph, a graph file in gpickle, a text file containing the graph stats and finally an image of the graph showing nodes, edges and the relations between them. There are 14 unique nodes and 63 unique edges. It is observed that not all graphs have the same edges and nodes, some may lack certain relations. Below is a list of the unique nodes and edges found from generating the provenance graphs.

|  |  |
| --- | --- |
| Nodes | |
| pipe | argv |
| process\_memory | path |
| iattr | socket |
| task | xattr |
| address | block |
| machine | file |
| shm | link |

|  |  |  |  |
| --- | --- | --- | --- |
| Edges | | | |
| read\_link | setattr | accept\_socket | removexattr\_inode |
| receive | free | listxattr | version\_entity |
| receive\_msg | clone | file\_lock | terminate\_proc |
| munmap | write | file\_rcv | write\_ioctl |
| memory\_write | connect | getxattr | receive\_unix |
| addressed | socket\_pair\_create | perm\_check | sh\_read |
| mmap\_private | exec | getxattr\_inode | setxattr\_inode |
| clone\_mem | link | exec\_task | named |
| unlink | setuid | ptrace\_read | listen |
| arg | memory\_read | ptrace\_read\_task | version\_activity |
| rename | removexattr | open | read\_ioctl |
| sh\_create | setxattr | accept | setpgid |
| sh\_attach | read | send | terminate\_tast |
| getattr | send\_unix | ran\_on | send\_msg |
| mmap | setattr\_inode | sh\_write | socket\_create |
| bind | connect\_unix\_stream | shmdt |  |

# Benign and Malicious Graph Comparison

Each benign graph and their malicious counterpart have different nodes and edges that can be used during model training to help assess which class each graph belongs to. Below are the pictorial forms of benign and malicious graphs for a side-by-side comparison. All graphs generated contained fine node and edge types provided by Camflow and whole system provenance capture granularity was selected.

## Visualising Graphs

|  |  |
| --- | --- |
| XSS Stored | Message Board Post |
|  |  |

|  |  |
| --- | --- |
| XSS Reflected | Completing a Questionnaire |
|  |  |

|  |  |
| --- | --- |
| XSS DOM | Querying a Webpage |
|  |  |

|  |  |
| --- | --- |
| Command Injection | Pinging Local Host |
|  |  |

|  |  |
| --- | --- |
| SQL Injection | Database Entry |
|  |  |

|  |  |
| --- | --- |
| Brute Force | Login |
|  |  |

## Node and Edge Comparison

### Between Same Graph Types

To find similarities and differences in the substructures of the graphs, the node and edge overlap for graphs within the same class and their malicious counterparts were calculated. The results listed below show that almost all graphs share a large overlap of nodes. Edge overlap is significantly lower within the same graph types showing less than 10% compared to same types of graphs.

Comparing MESSAGE Graphs

Node Overlap: 88.50%

Edge Overlap: 4.26%

Comparing SUBMIT Graphs

Node Overlap: 93.66%

Edge Overlap: 5.86%

Comparing QUERY Graphs

Node Overlap: 65.74%

Edge Overlap: 2.48%

Comparing PING Graphs

Node Overlap: 83.21%

Edge Overlap: 6.21%

Comparing DATABASEENTRY Graphs

Node Overlap: 58.02%

Edge Overlap: 5.32%

Comparing LOGIN Graphs

Node Overlap: 90.79%

Edge Overlap: 5.99%

Comparing XSSSTORED Graphs

Node Overlap: 98.56%

Edge Overlap: 4.90%

Comparing XSSREFLECTED Graphs

Node Overlap: 97.29%

Edge Overlap: 5.82%

Comparing XSSDOM Graphs

Node Overlap: 96.56%

Edge Overlap: 4.37%

Comparing SQLINJECTION Graphs

Node Overlap: 89.15%

Edge Overlap: 5.48%

Comparing COMMANDINJECTION Graphs

Node Overlap: 55.02%

Edge Overlap: 3.28%

Comparing BRUTEFORCE Graphs

Node Overlap: 62.94%

Edge Overlap: 3.34%

### Between Graph Counter Types

When comparing node and edge overlaps between benign graphs and their equivalent malicious graph, it is noted that node overlaps are significantly lower compared to our previous results, while edge overlaps remain relatively low indicating small differences.

Comparing XSSSTORED and MESSAGE Graphs

Node Overlap: 94.19%

Edge Overlap: 4.37%

Comparing XSSREFLECTED and SUBMIT Graphs

Node Overlap: 49.29%

Edge Overlap: 3.32%

Comparing XSSDOM and QUERY Graphs

Node Overlap: 34.10%

Edge Overlap: 2.64%

Comparing COMMANDINJECTION and PING Graphs

Node Overlap: 68.35%

Edge Overlap: 4.18%

Comparing SQLINJECTION and DATABASEENTRY Graphs

Node Overlap: 51.68%

Edge Overlap: 4.28%

Comparing BRUTEFORCE and LOGIN Graphs

Node Overlap: 53.74%

Edge Overlap: 3.11%

# Graph Neural Networks

## Overview

Recent trends have shown the gradual rise in Graph Neural Networks (GNNs) being used for more than academic research and being deployed more for real-life solutions. Deep learning algorithms that have the ability to extract high-level feature data from datasets have previously been tailored to consider structured, grid-like data such as the 2-dimensional grids of pixels in images and the 1-dimensional sequence of text while largely ignoring graph data [[site](https://www.assemblyai.com/blog/ai-trends-graph-neural-networks/)]

Convolutional Neural Networks (CNNs) extract local features and construct highly expressive representations which have led to breakthroughs in machine learning areas. However, the drawback of CNNs is that they only operate on regular Euclidean data such as images which are 2-D grids and text which are 1-D sequences [[paper](https://arxiv.org/pdf/1812.08434)]. This is not applicable to graphs due to their irregular structure and non-Euclidean data.

GNNs function by utilising a method called message passing, where data embedded within each node of the graph aggregates and updates its data based on its neighbouring nodes. Next, the input layer takes in the graph data, which is then processed by the hidden layers, and finally, the output layer classifies the node, edge, or graph according to the goal of the model. In between layers, similar to CNNs, rectified linear unit (ReLU) activation functions are used to introduce a nonlinear property to the model [[site](https://www.techtarget.com/searchenterpriseai/definition/graph-neural-networks-GNNs)]. It is worth noting that some GNN variants, including RGCNs, incorporate additional mechanisms like normalization or attention layers to enhance the model’s expressive power when working with more complex graph data.

## Relational Graph Convolutional Model

The provenance data gathered in the previous stages of this project has shown that the graphs contain different types of edges and nodes. These are known as heterogeneous graphs as opposed to homogeneous graphs, where nodes and edges are of the same type. A Relational Graph Convolutional Model is used to learn the representation vectors of the nodes in the graph and is better suited for use with heterogeneous graphs compared to Graph Convolutional Networks (GCNs), which mainly focus on homogeneous graphs [[paper](https://arxiv.org/html/2404.13079v1#:~:text=The%20difference%20between%20the%20RGCN,for%20each%20type%20of%20edge.)].

RGCNs extend GCNs by using separate learnable parameters for each edge type, allowing them to better capture the relationships in heterogeneous graphs. For each edge type, the RGCN applies a relation-specific weight matrix, which means that each relationship in the graph contributes uniquely to the embeddings of the connected nodes. This makes RGCNs particularly effective for modelling graphs with complex relational structures like the provenance graphs in this project.

## Node Feature Comparison

To learn more about the substructures within the graph, an RGCN process was applied to each graph in order for the nodes to aggregate and update their embeddings based on their neighbours. A two-layer RGCN with a ReLU activation function between them was used to update each graph's node embeddings, which were subsequently compared with each other. Each node in each graph was initialised with a random feature vector with 64 features, and a simple Euclidean distance comparison was used for each node type that both graphs had in common.

The results indicated that while most node types’ embeddings do not differ, some of them, particularly ‘machine’, ‘file’, ‘process\_memory’, ‘socket’, and ‘task,’ show significant differences between benign and their malicious counterpart graphs. These differences may stem from the distinct roles these node types play in the context of malicious activity. For instance, nodes representing malicious files or processes may exhibit unusual relationships or features due to their involvement in suspicious behaviours.

By identifying these differences, the analysis not only highlights the most discriminative node types but also provides insights into the underlying structure of malicious graphs. These insights can inform the design of more targeted and interpretable models in future iterations of this work.

Comparing node features of XSSSTORED and MESSAGE Graphs

Feature difference for node type 'address': 0.0027

Feature difference for node type 'file': 3.4334

Feature difference for node type 'iattr': 0.0040

Feature difference for node type 'link': 0.0114

Feature difference for node type 'machine': 11.2662

Feature difference for node type 'path': 0.0142

Feature difference for node type 'pipe': 0.0100

Feature difference for node type 'process\_memory': 2.2537

Feature difference for node type 'socket': 1.5990

Feature difference for node type 'task': 3.9144

Comparing node features of XSSREFLECTED and SUBMIT Graphs

Feature difference for node type 'address': 0.0000

Feature difference for node type 'file': 4.2199

Feature difference for node type 'iattr': 0.0028

Feature difference for node type 'link': 0.0727

Feature difference for node type 'machine': 11.2509

Feature difference for node type 'path': 0.0396

Feature difference for node type 'pipe': 0.0118

Feature difference for node type 'process\_memory': 2.9368

Feature difference for node type 'socket': 1.6073

Feature difference for node type 'task': 3.1063

Comparing node features of XSSDOM and QUERY Graphs

Feature difference for node type 'file': 3.7962

Feature difference for node type 'link': 0.0000

Feature difference for node type 'pipe': 0.0220

Feature difference for node type 'process\_memory': 2.2925

Feature difference for node type 'socket': 0.9188

Feature difference for node type 'task': 1.8679

Comparing node features of COMMANDINJECTION and PING Graphs

Feature difference for node type 'address': 0.0039

Feature difference for node type 'file': 3.7627

Feature difference for node type 'iattr': 0.0034

Feature difference for node type 'link': 0.0177

Feature difference for node type 'machine': 11.2747

Feature difference for node type 'path': 0.0472

Feature difference for node type 'pipe': 0.0182

Feature difference for node type 'process\_memory': 2.6366

Feature difference for node type 'socket': 1.8636

Feature difference for node type 'task': 3.4637

Comparing node features of SQLINJECTION and DATABASEENTRY Graphs

Feature difference for node type 'address': 0.0000

Feature difference for node type 'file': 4.2610

Feature difference for node type 'iattr': 0.0000

Feature difference for node type 'link': 0.0232

Feature difference for node type 'machine': 11.2493

Feature difference for node type 'path': 0.0154

Feature difference for node type 'pipe': 0.0122

Feature difference for node type 'process\_memory': 2.7710

Feature difference for node type 'socket': 1.6098

Feature difference for node type 'task': 2.7836

Comparing node features of BRUTEFORCE and LOGIN Graphs

Feature difference for node type 'address': 0.0000

Feature difference for node type 'file': 3.8742

Feature difference for node type 'iattr': 0.0012

Feature difference for node type 'link': 0.0888

Feature difference for node type 'machine': 11.2491

Feature difference for node type 'path': 0.0125

Feature difference for node type 'pipe': 0.0100

Feature difference for node type 'process\_memory': 2.5919

Feature difference for node type 'socket': 1.9825

Feature difference for node type 'task': 3.6577

# Graph Classification

## Model Architecture

### Relational Graph Convolutional Network (RGCN)

The RGCN is responsible for performing message passing and learning node embeddings for each type of node in the graph. For this project, two layers of graph convolution were applied to the graph’s node embeddings. Between the two layers, a ReLU activation function was used to introduce non-linearity and enhance the model’s ability to capture complex relationships within the graph. The RGCN leverages the graph's heterogeneity by applying separate convolution operations for each edge type.

### Heterogeneous Graph Classifier

The graph classifier module is designed to classify the graph as a whole, predicting whether it is benign or malicious based on the aggregated node features. The process is as follows:

1. The RGCN generates updated node embeddings for each type of node in the graph after the message-passing steps.
2. These embeddings are aggregated into a single graph-level representation using mean pooling across each node type.
3. The aggregated representations from all node types are summed together to form a holistic graph embedding.
4. A fully connected layer maps this graph embedding to the desired class label, where 1 represents a malicious graph, and 0 represents a benign graph.

## Parameter Tuning

### Feature Embedding Sizes

Experiments were conducted with node feature embeddings of sizes 32, 64, and 128. While attempting to use 256 embeddings, computational constraints resulted in out-of-memory errors, preventing these experiments from being completed.

### Epochs

The training process was tested with 5, 10, and 20 epochs to identify the optimal number of iterations required for the model to converge effectively and produce accurate results. The trade-off between training time and accuracy was considered when selecting the best configuration.

## Results

-Insert Table for 32, 64, 128 10 epochs-

-Insert Table for 64, 20 epochs-

-Insert Table for all graphs acc, prec, recall, f1-

# Conclusion

# Further Improvements

# References