Generating Custom Real-World Activity Data to Train an Artificial Intelligence Cloud Cybersecurity Model

A Technical Report

Submitted to the Department of Computer Science

Presented to

The Faculty of the

School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science in Computer Science

By

Claire Williams

May 2, 2024

Advisor

Rosanne Vrutgman, Department of Computer Science

Generating Custom Real-World Activity Data to Train an Artificial Intelligence Cloud Cybersecurity Model

CS4991 Capstone Report, 2024

Claire Williams
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
cmw6zug@virginia.edu

ABSTRACT

AWS Detective is a cloud security service that allows the user to visualize and analyze their activity to detect potential security threats and the detection algorithms need training data to develop and improve. To facilitate this, I created a program to generate custom activity data to model security threats and a lack thereof on demand. Security engineers can then use this tool to generate training data for the algorithm. I implemented a program that could read various formats of JSON inputs and mock the activity data. The major considerations in creating this tool were embedding specific attacks in the activity and creating data that appeared real at a high volume. In terms of future work, I only implemented the most important/common actions, and some activity still cannot be generated with this method. The tool can continue to be developed to mock more sophisticated scenarios. Also, the front end of the tool can be made more usable for the security engineers.

1. INTRODUCTION

AWS Detective is a cloud service that processes a user's activity data into a graph-like data structure and uses this graph to simplify the security investigation process. In this graph, the nodes are called "entities" and they are actors in the network activity, such as individual users, IP addresses, EC2 instances (computing resources), and S3 buckets (storage resources), for example. The edges between the entities are called "relationships" and those are actions in the activity. Some common examples are "put file in bucket" as a relationship that connects user and bucket entities or "user assumed role" as a relationship connecting user and role entities. Within this graph structure, each entity and relationship has metadata associated with it like "time created" or "arn (Amazon Resource Name)". Some of the metadata is simple, like a text label or integer number. Some metadata are more complicated like time-series data of bytes transmitted or a running total count that updates. Entities and relationships have various metadata based on whatever information is relevant to them. Therefore, the graph of these entities and relationships provides a very holistic view of all of the activity in an AWS account.

When there are security vulnerabilities or potential cyberattacks, this graph data structure helps investigators trace the attack and identify corrupted resources or bad actors. AWS Detective runs algorithms against this graph data structure to help identify points of interest and highlight activity related to an attack. To refine these algorithms, the security engineers working on them need training activity data. However, due to security concerns, they cannot use real customer data. The goal of my project was to generate custom activity graphs on demand for use by security engineers.

This activity needed to range from simple to extensive to represent various types of customers, such as individual accounts versus large corporate accounts. Engineers also needed to be able to embed specific attack sequences in the activity to see if their algorithms could recognize the relevant activity. Therefore, some of the activity should be of a large volume and not be overly specified, whereas some small volume of activity should be able to be exactly specified. The activity also needed to be as similar to real data as possible to prevent the algorithms from overfitting. For example, if the algorithm learns to identify simulated activity versus real activity, it may flag simulated activity in training because it knows it is fake, rather than actually identifying the vulnerability. This would make the algorithm's performance appear much better than it actually is.

In summary, the main goal of the tool is to synthesize data both on a large and general scale and a small and specific one, as realistically as possible. This was an ambitious goal, so this tool was intended to lay a foundation rather than fully achieve the goal in the timeframe.

2. RELATED WORKS

Previous work in generating training data for artificial intelligence (AI) models stresses the importance of avoiding overfitting. Overfitting is when a model performs significantly better on the data it was trained on versus new testing data. This is because the classification model learned something about the data it was trained on rather than the problem it was seeking to address. Previous research has presented various validation techniques to test for overfitting. One study showed that machine learning models can predict remission of Crohn's disease better than multivariate logistic regression models when the proper validation techniques are used. Researchers detail techniques such as cross-fold validation and hyperparameter turning that help avoid over-fitting and poor generalization [1]. This provided some worthwhile considerations for creating training data.

Another group of researchers did a comparative study of machine learning techniques in cybersecurity. They found that machine learning can provide enhanced security and help detect zero-day attacks with less human intervention. These researchers also suggest some valuable future areas of research [2]. This result is promising as it reinforces AWS Detective's mission.

3. PROJECT DESIGN

This tool began with an extensive design process, as there was no previous existing work. The following sections delve into major considerations chronologically.

3.1 Divide Problem into Description-to-JSON and JSON-to-Graph

Since each element in this graph can contain a significant amount of metadata, and this project was intended to build the foundation of a more sophisticated tool, we approached this design with extensibility as a major consideration. Early on, we decided to split the problem of turning a description into a graph into two main parts: creating a detailed JSON from a description and creating a graph from a detailed JSON. These two subproblems are distinct and all-encompassing.

The first subproblem is creating a graph from a detailed JSON. This means that all of the information contained in the graph is listed in the JSON and this process parses the JSON and creates it in the system. This JSON must be able to represent all possible manifestations of the activity data, as it provides the foundation of the tool. The second subproblem is generating these descriptive JSONs from a human-understandable conception of what a graph should look like. This problem is much less defined. Separating the problem in this manner disconnects the design of this tool from many of the technical constraints such as the existing definitions of the data.

3.2 Generating Fully Defined Entities and Relationships

We started with creating a graph knowing exactly what it should contain. To do this, we reused testing infrastructure that hardcoded some graphs to do more design work and less infrastructure work early on. This part of the program would read a large JSON and put the information into a graph.

Next, we worked on creating these graphs in a way that abstracts away as many of the details as possible. For example, each entity has many fields of metadata that need to be explicitly established, so we wanted to introduce defaults for these values such that when they were created en masse we would not have to worry about smaller details. We did this using method overloading.

Method overloading is when there are many methods with the same name but different sets of arguments. The program runs the method declaration with the correct set of provided arguments. In this case, for example, we create the createEntity method which had a variety of arguments such as:

- createEntity(String entityType)
- createEntity(String entityType, String idPrefix)
- createEntity(String entityType, Feature[] features)
- createEntity(String entityType, String idPrefix, Feature[] features)

This method existed for every permutation of possible information about an entity such that one could call the method with whatever information they wanted specified and the rest would be auto-populated. This was more complicated than it seems because different types of entities can have different fields so the program has to do a reverse lookup to find an appropriate default value, however, this approach hides these idiosyncrasies at a higher level of abstraction. The same concept applied to relationships. All of this information was kept track of and outputted in a JSON fully specifying a graph.

For the first step, we reused existing infrastructure to create graphs that were entirely specified. Then, we used method overloading to establish default values for everything and abstract away many of the details.

3.3 Generating Entities and Relationships with Simple Features in Bulk

The next step was creating these entities and relationships in bulk. This step reused the previous method overloading such that the user could still specify whatever information about a group of entities and then add a count of how many they wanted. This also allowed for creating relationships en masse and only specifying the type of entities that should be connected.

This was only implemented for simple features; if one wanted a more complicated feature, it had to be created individually. Extending this for complex features is an important element of future work. Rather than directly creating a graph from these specifications, the program would create a detailed JSON fully describing the graph, and then create the graph from that JSON.

3.4 Injecting Specific Attack Patterns

Another major requirement of this tool was to be able to inject specific attack scenarios into the graph to see if the algorithms could identify them. Since cyberattacks follow certain methodologies, security engineers have specific scenarios that are examples of types of attacks. The program can add pieces of activity one by one but one can imagine a situation where security engineers are reusing specific attack scenarios. Therefore, we added the capability to combine JSONs or inject a specific JSON into a larger JSON. This way, engineers can reuse attacks by saving them in files and quickly injecting them.

3.5 User Interface to Combine Input Types into Single Activity Output

After creating the main functionality of the tool, we worked on a user interface. The program has three main capabilities: creating activity piece by piece with any degree of specification, creating activity in larger pieces by specifying counts of repeated elements, and injecting specific activity from other JSONs. For the user interface, we settled on a sandbox-like model where users can repeatedly do any of the three main types of input for as long as they want. Then, when the user is finished adding to the graph, they can export all of the information to a single JSON file. Then, they can input that JSON into the graph generator.

4. RESULTS

The current implementation of the tool provides the foundation for future work and serves as a proof of concept for this idea. The tool was designed with extensibility in mind so the problem has been reduced from making custom graphs on demand to making custom JSONs on demand. Additionally, the overloaded method headers provide tools to make more sophisticated activity later on and do more processing on the default values. The current state of the tool is that it can generate graphs and data at scale and model specific attacks but it cannot generate a lot of realistic-looking activity.

5. CONCLUSION

This project was implemented as the foundation for automating test data production. This is important because test data is necessary to train AWS Detective's cybersecurity algorithms. The tool was built with extensibility in mind so, as the platform evolves, the tool will hopefully only need simple modifications to be compatible. This will indirectly benefit customers as it will help improve the cybersecurity algorithms of AWS Detective. This will provide the customers with improved threat detection and more relevant activity to investigate these threats.

6. FUTURE WORK

This tool serves as a proof of concept, and there is still future work to be done before it can be used to train algorithms. The tool's capabilities should be expanded to model complex features as well. For example, the tool cannot automatically create time series features, such as how many bytes of data are transmitted over time. Additionally, creating features en masse can currently only be done randomly, but not modeled after typical real-world activity. Therefore, the algorithms may overfit on this random data. Also, as the tool develops, continuous testing must be done concurrently to detect overfitting and other issues that the test data may propagate to the algorithms.

REFERENCES

[1] Charilaou, P., & Battat, R. (2022). Machine learning models and over-fitting considerations. *World journal of gastroenterology*, *28*(5), 605–607. https://doi.org/10.3748/wjg.v28.i5.605

[2] Wazid, M., Das, A. K., Chamola, V., & Park, Y. (2022). Uniting cyber security and machine learning: Advantages, challenges and future research. *ICT Express*, *8*(3), 313-321. https://doi.org/10.1016/j.icte.2022.04.007