Mecha: A Neural-Symbolic Open-Set Homogeneous Decision Fusion Approach for Zero-day Malware Similarity Detection

Christopher Molloy*, *Student Member, IEEE*, Jeremy Banks*, Steven H. H. Ding*, *Member, IEEE*, Furkan Alaca*, Philippe Charland†, and Andrew Walenstein‡

***Abstract*—With increasing numbers of novel malware each year, tools are required for efficient and accurate variant matching under the same family, for the purpose of effective proactive threat detection, retro-hunting, and attack campaign tracking. All of the state-of-the-art Deep Learning (DL) approaches assume that the incoming samples originate from known families and incorrectly identify novel families. Additionally, most of the existing solutions that leverage the Siamese Neural Network architecture either rely on pair-wise comparisons or computationally expensive preprocessing steps that are not scalable to a real-world malware triage volume requirement.**

We propose a different route, Mecha, a Neural-Symbolic Machine Learning (ML) system for malware variant matching and zero-day family detection. Mecha is comprised of an embedding network trained in two different scenarios for byte string embedding and an open-set approximate nearest neighbour algorithm for variant matching and zero-day detection. Our embedding network uses triplet loss for embedding generation and reinforcement-based Expectation Maximization (EM) learning for full deployment optimization. We conduct multiple in-sample and out-of-sample experiments to demonstrate the model's generalizability toward novel variants and families. We also show that Mecha can detect samples outside the known set of malware samples with an accuracy greater than 0.990.

***Index Terms*—Deep Learning, Reinforcement Learning, Cybersecurity, Malware Analsys.**

I. Introduction

With rapid advances in Artificial Intelligence (AI) and DL in recent years, learned systems are being applied to aid humans in all aspects of life. Areas that require large amounts of data processing have seen greater success with applications of AI. One such area is cybersecurity. In the cybersecurity task of malware analysis, governments and corporations must search through millions of files entering their network each year for malware. According to the AVTEST Institute, there have been 70,687,826 new unique malware samples cataloged for Windows systems alone in 2022[1]. Human investigation is impossible at this scale, and traditional signature-based methods can be evaded through packing and other obfuscation techniques [1]–[3].

*C. Molloy, J. Banks, S. H. H. Ding, and F. Alaca, are with the School of Computing, Queen's University, Ontario, Canada K7L 2N8. e-mails: {chris.molloy, jeremy.banks, steven.ding, furkan.alaca}@queensu.ca, †P. Charland is with the Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada. E-mail: philippe.charland@drdc-rddc.gc.ca, ‡ A. Walenstein is with Security Research and Development at BlackBerry. E-mail: awalenstein@blackberry.com.

Manuscript received xxx; revised xxx; accepted xxx

[1]https://portal.av-atlas.org/

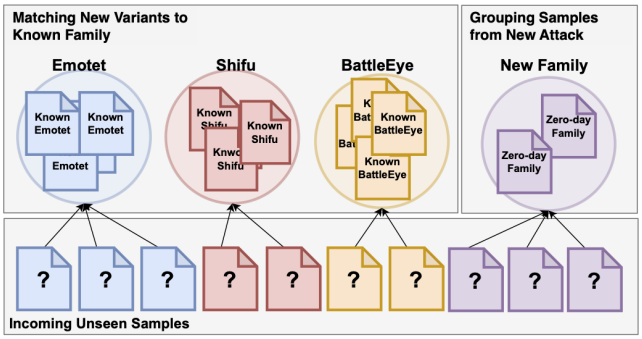

Fig. 1. Incoming malware typically consists of new variants of known families or samples associated with zero-day attack campaigns. A classification-based approach cannot handle samples for zero-day attacks or new malware families. Using a similarity analysis system, incoming samples are analyzed and compared to samples from known families. If samples are not within a predefined boundary, it is likely a zero-day family.

The task of malware analysis is not simply finding if a file has malicious intent or not. Challenges within the space of malware analysis are family detection, similarity analysis, and zero-day family detection [3], [4]. With the vast amounts of unique malware samples, they are categorized into families. A malware family is a set of malware that share a distinct sample of malicious code [5]. The first sample in a family is an unseen and unique piece of malicious software. All other samples within the family are variants, iterations of the original sample with minor differences [5]. Following that, a variant is a modification of the original code with minor differences; a malware sample can only be categorized into a single family. It is important to note that varying definitions of malware family exist, including definitions that base a family on similar behaviour [6]. Conventional methods of family detection aim to categorize new samples into a family based on a known knowledge space [4], [7]–[12]. An emerging area of research is malware similarity analysis, where methods are proposed for evaluating the similarity between malware samples, allowing for a more nuanced analysis [13]–[15]. By providing a similarity score between each sample, Cyber Threat Intelligence (CTI) specialists can map the evolution of malware variants for attack campaign monitoring. Another possible application of malware similarity analysis is zero-day family detection. Zero-day family detection is the challenge of finding novel distinct samples of malicious code to flag for human investigation. With discrete set family classification,

there does not exist a mechanism for signaling if an incoming sample is outside the set of families. With samples from zero-day families having unknown signatures, structures, and intentions, a comprehensive CTI system requires zero-day family detection for network defence. A visualization of these two problems can be seen in Fig. 1. Similarity hashing, another popular solution, uses a rolling method to generate a hash for malware samples and uses an editing distance for finding the similarity between files [16]–[18].

One DL approach to malware similarity analysis is applying Siamese Neural Networks. Siamese Neural Networks learn in either a twin or triplet meta-learning method for direct classification, similarity scoring, or similarity embedding generation. Methods have shown the ability to match similarity between malware samples on the basis of malware family [13]–[15], but only [13] generated similarity embeddings for Windows malware. Through leveraging a heterogeneous set of malware descriptors for each sample in [13], Molloy *et al*. designed an embedding network for malware sample storage and family detection.

Although [13] has shown great success at matching samples through measuring the distance between embeddings, there are challenges within the domain of malware similarity analysis that have not been addressed. One such challenge is accurately matching malware samples that have been explicitly modified for evasion [19], [20]. In a recent study, we have shown that appending benign bytes to the end of a malware sample is the best method for evading an ML-based malware classifier [19]. One potential solution against this method is by fragmenting incoming samples and performing similarity analysis on each fragment, then aggregating the fragmented results. A visualization of this possible solution compared to the current state-of-the-art can be seen in Fig. 2. Matching at the fragment level can help alleviate the issue of byte injection, as the decision-making process does not have to consider all fragments equally.

However, creating a solution that performs fragment-level matching is difficult. In contrast to current solutions, the input dimension to the Siamese network in a fragment-based solution is much smaller. From this, we found that networks performed well in the triplet training task, but when transferred to a deployed environment where incoming fragments were compared to known fragments for similarity analysis, performance decreased dramatically. To address this neighbour search challenge, we require training for neighbour search. Such training would simulate a deployment environment, where fragments are matched to their closest neighbour for similarity analysis (step 1 of our proposed solution in Fig. 2). This method requires a set of fragments pre-embedded by the network for training samples to search through (support set). It is important to note that this training can not optimize the model directly, because the support set used would contain the error of the model, making it only an approximation of the optimal. Therefore, an iterative training method that updates both the network and the support set is required for network optimization. A solution to this approach is Expectation Maximization (EM). EM is an iterative approach to finding the optimal parameters of a statistical model that

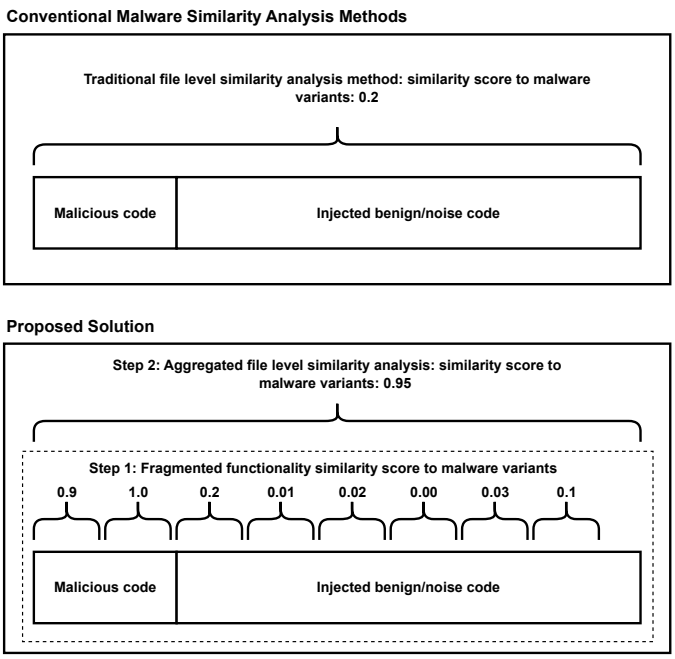

Fig. 2. A comparison of conventional malware similarity analysis tools to the proposed solution. In the event of a byte-appending attack, conventional methods have no mechanism for separating true malicious code from injected code. The proposed solution follows a two-step fragment and aggregate approach, allowing for accurate variant matching.

cannot be solved directly. By interpreting the search results on training samples as a probability distribution of families, a reward can be derived and trained through the network using a Reinforcement Learning (RL) paradigm. Then, the support set can be re-generated with the updated distribution parameter estimates.

We propose Mecha, a malware embedding and open-set family detection system that leverages multiple ML models. Mecha is divided into three sub-components: Mecha.emb, Mecha.gen, and Mecha.match. We propose Mecha.emb, a novel malware embedding network that embeds kilobyte (1,024) long byte strings of software byte code. This network aims to generate an embedding for a byte string that best represents the functionality of the kilobyte instead of an embedding that best represents the family of the sample. As well, we train our network on triplet pairs derived from malware and benignware to ensure a differentiation in byte string functionality. Unlike conventional solutions, such as similarity hashing, the Mecha.emb system embeds data into a hypersphere, which enables scalable approximated nearest neighbour search. To address the challenge of fragment matching, we propose a dynamic embedding nearest neighbour matching training gym, Mecha.gen. The Mecha.gen training gym conducts a nearest neighbour search on a changing support set for network generalization. Due to a direct loss being calculated from similarity search being non-differentiable, we deploy an RL scheme with EM for sequential search and update training. We show that training the embedding network in a simulated malware triage environment after initial training raises family matching accuracy, and decreases the distance between samples within

the same family. Finally, we build on the state-of-the-art approximate nearest neighbour algorithm [21] for open set family detection. An overview of the Mecha system can be seen in Fig. 3. We conduct multiple experiments on both in-sample and out-of-sample malware families to ensure our model is generalized and accurate at malware family detection. The main contributions of this paper are as follows:

- We design the first multi-ML system for malware variant matching that leverages the meta-learning structure of Siamese Neural Networks.
- We design an RL method for further training the embedding network for dynamic support set nearest neighbour search.
- We combine decision fusion of byte string variant matching from open-set nearest neighbour search for fast and accurate variant matching.
- We train Mecha on real-life malware samples and evaluate it on in-sample, and out-of-sample malware families in chronological order. The experiment shows that our model outperforms current state-of-the-art methods for malware variant similarity analysis in both known, and zero-day family detection.

The organization of this paper is as follows. First, we discuss the threat model for the Mecha system. Second, we discuss the Mecha.emb subsystem. Third, we discuss the Mecha.gen training method. Fourth, we discuss the Mecha.match system. Fifth, we discuss the experiments used to quantify the efficacy of the Mecha system. Sixth, we discuss related works. Seventh, we discuss possible future directions.

II. Threat Model

The Mecha system works as a family classification tool. Mecha can be incorporated into a CTI pipeline for classifying malware samples and showing similar samples, leading to the decision simultaneously. Due to the Mecha system being designed for classification, other tools are required for malware triage and detection. Mecha can be used to gain a deeper understanding of malware samples collected by CTI professionals. In a real-world deployment setting, Mecha can be used for classifying known malware into families. In a research setting, Mecha can be used to explore the similarity of zero-day samples within a family as well as family classification.

Additionally, since Mecha is designed for malware similarity analysis, incoming samples are assumed to be correctly classified as malware. In a typical CTI pipeline such as Assemblyline[2], the processes are distinct: starting from endpoint malware triage, progressing to server-side malware filtering, and finally ending in family classification and investigation through both static and dynamic analysis. These are separate stages with specific roles. The family labelling step assumes that prior stages in the pipeline are configured with an acceptable level of false positives and negatives. Such false positives are expected to undergo further investigation to either confirm attacks or refine the endpoint triage configuration to avoid similar cases in the future. Family labelling provides insights into the detected attacks, such as their tactics, purposes, and provenance. It builds on the foundational assumption that the false positive and the false negative levels from earlier stages are adequately managed, given the capacity of the security team and the risk tolerance level of the organization. Mecha's design is intentionally isolated from the preceding steps to maintain independence, as any other alternative solutions during this step. Its effectiveness does not hinge on the specific methods employed in earlier classification stages, but it acknowledges that false positives from those steps require investigation regardless. If an incoming sample has undergone significant modification for evasion by adding benign binary fragments, the worst-case scenario for the sample is being classified as 'unknown' and requiring further investigation. The Mecha system's advantage is its ability to classify malware as 'unknown'. This classification often indicates the possibility of a new attack family with distinct behaviours or tactics, which warrants further human investigation. Such cases are particularly valuable as they may reveal novel attack patterns, origins, or methodologies. When traffic is classified as 'unknown', CTI professionals at the Security Operations Center (SoC) are prompted to initiate an in-depth analysis. This process typically involves methods such as code analysis, extended sandboxing, and attack graph analysis to uncover the nature of the sample. The aim is to determine whether the sample belongs to an unrecognized malware family or represents the first instance of an entirely new family. The insights gained from these investigations are critical. If a new family is identified, it can be incorporated into Mecha's training sets to enhance its classification capabilities to trace emerging new attacks. Additionally, understanding the distinct behaviours and tactics of the unknown sample enables better preparation against similar threats in the future. By enabling this escalation and analysis process, Mecha not only aids in labelling existing malware families but also plays a pivotal role in the discovery and characterization of emerging threats.

[2]Assemblyline Malware Processing System

III. Mecha.emb

Mecha.emb is a deep convolution network for byte string embedding. Given a string of $1{,}024$ bytes from a software executable, Mecha.emb generates an embedding that represents the byte string in Euclidean space.

A. Embedding Modality: Byte String

The input to the Mecha.emb network is a byte string of length $1{,}024$ extracted from the malware. A length of $1{,}024$ was chosen to balance the amount of functionality that can be modelled by the byte string, with the input size to the neural network. As well, the batch mode of computation done within CPUs and GPUs was considered. A byte string is a vector of integer values with the range $[0, 255]$ with each value representing a byte of binary code. For binary code, we refer to all bytes in an executable. Byte strings were chosen as the input modality to the Mecha.emb system, due to the speed and accuracy of extraction. Many ML-based systems designed for malware analysis require each sample to be decompiled prior to analysis. Decompilation is not a deterministic process, and

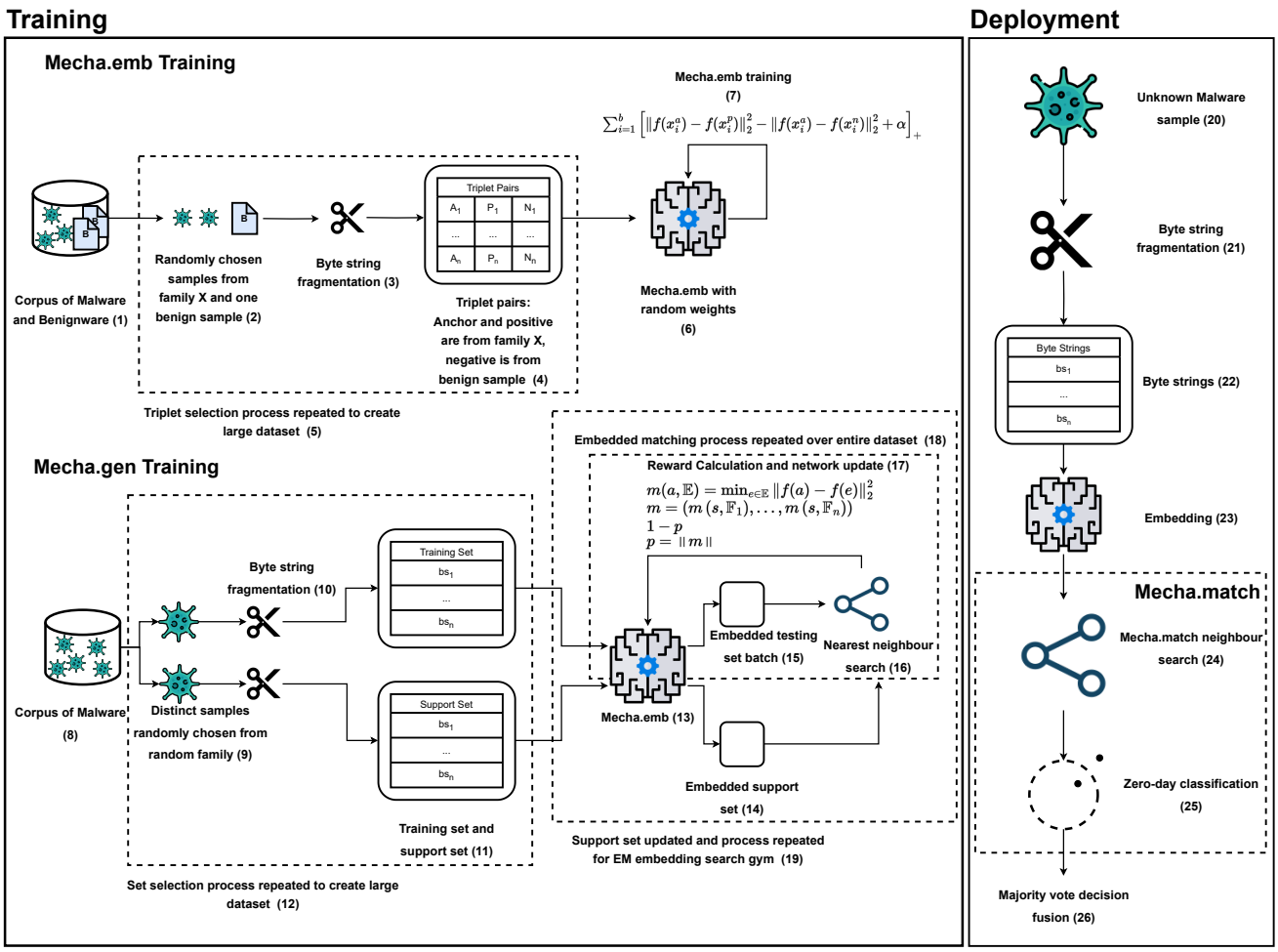

Fig. 3. Overview of the Mecha system. (1) a corpus of malware and benignware is collected. (2) two malware variants from the same family, as well as a benign sample, are randomly chosen. (3) the files are fragmented into byte strings of length 1,024. (4) a set of triplet pairs is generated from the fragmented byte strings. (5) this process is repeated to create a large dataset of different family triplet pairs. (6) the Mecha.emb network is initialized with random weights. (7) the Mecha.emb network is trained with a triplet loss. (8) a corpus of malware is collected. (9) two different variants from the same family are chosen from a random family. (10) both samples are fragmented into their byte strings. (11) a training set and support set is generated from the byte strings. (12) the set selection process is repeated to create large and diverse training and support sets. (13) the pre-trained Mecha.emb is ready for training. (14) the support set is embedded by the Mecha.emb network. (15) a batch of the training set is embedded by the Mecha.emb network. (16) a neighbour search is conducted on each sample in the training batch to the support set. (17) a reward is calculated for the batch matching and propagated through the Mecha.emb network. (18) the batch neighbour search is conducted over the entire training set. (19) at the end of training from the entire set, the support set is embedded with the updated Mecha.emb, and training continues. (20) an unknown malware sample is introduced. (21) the unknown sample is fragmented into byte strings. (22) the byte strings are organized for embedding. (23) the Mecha.emb network trained through Mecha.gen embeds the unknown byte strings. (24) the Mecha.match system finds the nearest neighbours to the incoming byte strings. (25) zero-day classification is done on the unknown byte strings. (26) a majority vote decision fusion is conducted to classify the unknown sample.

many parameters within decompilation software may greatly affect the resulting source code that is used to analyze the sample. Byte strings can be read directly from the executable file, removing any non-deterministic process from the embedding system.

B. Model Structure

Prior work on malware similarity analysis through Siamese networks has generated either embeddings or similarity scores based on information from the entire sample [7], [8], [13]–[15], [22]. No work has been done in the malware family detection space by using byte strings of the raw executable as the input for an embedding network. This differentiates the motivation between the Mecha.emb network and other Siamese networks used for malware family detection. Whereas prior work aims to train a network for finding the similarity between entire malware samples, the motivation for Mecha.emb is reducing the effect of minor changes made to malware by creating multiple embeddings for a single sample.

The structure for Mecha.emb is based on the FaceNet architecture proposed in [23]. The structure of FaceNet is a collection of two-dimensional convolution, pooling, and normalizing layer blocks. Due to the input of Mecha.emb being one-dimensional byte strings, the two-dimensional convolution layers were replaced with one-dimensional convolution. The FaceNet architecture has seen great success in the domain of image embedding [23]. Similar networks have also seen success in the malware similarity analysis domain without the use of normalization layers throughout the network [14].

TABLE I
STRUCTURE OF THE MECHA.EMB DEEP CONVOLUTION NETWORK.

Layer	Size-in	Size-out	Kernel	Params
embedding	1024	1024×8		2048
conv1a	1024×8	897×64	128	65600
mpool	897×64	224×64		0
bnorm	224×64	224×64		256
conv1b	224×64	193×32	32	65568
mpool	193×32	96×32		0
bnorm	96×32	96×32	16	128
conv1c	96×32	81×32		16416
mpool	81×32	20×32		0
bnorm	20×32	20×32		128
flatten	20×32	640		0
L2	640	640		0
Total				150144

Although Mecha.emb works within the domain of malware analysis, Mecha.emb normalizes data throughout the architecture due to the embedding nature more resembling Facenet than other networks in malware similarity analysis. Although it is typical to use sequence-based network layers on one-dimensional byte strings, such as a Long Short Term Memory or Gated Recurrent Unit layer, one-dimensional convolution layers have seen success in the area of malware detection with the added benefit of reduced runtime [24]. The structure of the Mecha.emb network can be seen in Table I. The Mecha.emb architecture has fewer convolution blocks than previously discussed work. This was chosen due to the input dimension of the Mecha.emb network being significantly smaller than other networks discussed. Further modifications have been made to the model structure to fit the malware embedding challenge. Through empirical testing, we found using max-pooling layers with different dimensionality aided in generating a more effective malware embedding. Using different pooling sizes allows for control of the size of the final embedding of a sample without significantly increasing the depth (as well as runtime) of the embedding model. Also, any spatial inconsistencies are handled by truncating the input.

C. Triplet Loss

We represent an embedding of byte string x as $f(x) \in \mathbb{R}^d$ where $f(\cdot)$ is Mecha.emb, and d is the output dimension of the Mecha.emb network. As described in Table I, the output dimension of $f(\cdot)$ is $d = 640$. The training of the Mecha.emb network is done using a Euclidean triplet loss derived in [23]. We will now describe the loss in detail.

The Mecha.emb network does not learn from any ground-truth malware-to-family tuples, but compares its output of different samples for deriving a loss. The methodology of deriving loss from a distance between three samples is conventional for embedding networks trained in a triplet paradigm [23]. This loss is related to the distance between the three samples, two from the same family and one from a different family. Although a lower loss value implies that the network embeds samples from the same family closer to one another, it does not explicitly indicate possible error in classifying malware samples into families. For a loss calculation of Mecha.emb, three byte strings are required. The first two are byte strings of length $1{,}024$ from different samples in the same malware family at the same starting index. These two samples are the *anchor* and the *positive* samples, denoted as x^a and x^p respectively. The third byte string is a random byte string from a benign sample, and serves as the *negative* sample denoted as x^n.

Using a benign sample as the negative for the training is a nontrivial decision. In prior works on malware family detection using similarity loss, the motivation of the model training is to find the similarity in malware families, whereas the motivation of training Mecha.emb is to find the similarity of byte string functionality. Using benign software as the *negative* sample ensures differing functionality between the *negative* and the *positive*.

Triplet loss is calculated by finding the difference in distance between the *anchor*, *positive*, and *negative*. The motivation for this loss is training the Mecha.emb network to output embeddings close to one another in Euclidean space if they have similar functionality, and apart from one another if they have differing functionality.

The set of a single *anchor*, *positive*, and *negative* is known as a triplet pair, $\mathbb{T}$. Mecha.emb is trained on the loss of a batch of triplet pairs of length b at each train step. We denote our batch as $\mathbb{B} = \{\mathbb{T}_1, \mathbb{T}_2, ...\mathbb{T}_b\}$ where $\mathbb{T}_i = \{x_i^a, x_i^p, x_i^n\}$. For a single train step we calculate our loss as $L =$

$$\sum_{i=1}^{b} \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right]_+ \quad (1)$$

where α is a margin enforced between positive and negative pairs. Due to the distance function in the loss being unbounded (the range of Euclidean distance is $[0, \infty)$), learning on all triplet differences slows down convergence [23]. The constant α ensures only triplet differences that are within the specified margin are trained through the network. For Mecha.emb training, we use an $\alpha = 0.2$. We chose a margin value of 0.2 due to its success in [23].

In experiments, we train the Mecha.emb network using the Adam Stochastic Gradient Descent method [25]. We train the Mecha.emb model with a learning rate of 0.0001 for 10 epochs. These values were chosen from an empirical evaluation of model training (observing training efficiency with differing learning rates and iterations).

D. Triplet Selection

As described above, we use malware and benignware for creating our triplet dataset for training the Mecha.emb network. We will now describe the process for generating the dataset. For triplet selection, files are chosen with replacement from a corpus of malware and benignware. Three files are chosen from the corpus for a single triplet pair. The files used for the *anchor* and *positive* are chosen from the same malware family without replacement. The *negative* is chosen randomly from the set of benignware. We define each file as a set of byte strings. We define each set to contain all non-overlapping byte strings of length $1{,}024$ contained in the chosen file. This method removes the final byte string of each file that is less than length $1{,}024$. This is done to avoid any negative impact on similarity training caused by padding that would be necessary

on the final byte string to extend it to 1,024 bytes. For the *anchor*, *positive*, and *negative* we define the byte string vectors as $\boldsymbol{a}$, $\boldsymbol{p}$, and $\boldsymbol{n}$ respectively.

For the triplet pair, an index q is randomly chosen as the starting index, subject to the following constraints:

$$q \equiv 0 \mod 1{,}024. \tag{2}$$

$$q < \min\left(\dim\left(a\right), \dim\left(p\right), \dim\left(n\right)\right) - 1{,}024, \tag{3}$$

These constraints omit the byte string at the end of each file. A triplet pair is the byte strings of length 1,024 from $\boldsymbol{a}$, $\boldsymbol{p}$, and $\boldsymbol{n}$ at index q. The data for training and validating the Mecha.emb network is generated using this method. For a set of malware and benignware, a large number of unique triplet pairs can be sampled for training. We will now describe the maximum number of triplet pairs that we can generate from a set of malware and a set of benignware.

First, we define the software cutting function $c : \mathbb{N}^{\alpha} \rightarrow \mathbb{N}$ where α is arbitrary that has an input of a software sample, and outputs the number of byte strings that can be generated from the sample. We define c as

$$c\left(l\right) = \dim\left(l\right) - \dim\left(l\right) \mod 1{,}024 \tag{4}$$

where l is a software sample. Given a set of malware $\mathbb{M} = \{\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_m\}$ of size m where $\mathbb{F}_j$ is a set of malware samples in family j, and a set of benignware $\mathbb{S} = \{b_1, b_2, \ldots, b_k\}$ of k benign software samples. For an arbitrary malware family $\mathbb{F}_{\text{J}} = \{f_{j,1}, f_{j,2}, \ldots, f_{j,p}\}$ with p samples, the number of triplet pairs that can be made for arbitrary malware sample $f_{j,r}$ can be computed by

$$\sum_{y=1}^{k}\left[\left(\sum_{x=1}^{p}\min\left(c\left(f_{j,r}\right), c\left(f_{j,x}\right), c\left(b_y\right)\right)\right) - c\left(f_{j,r}\right)\right]. \tag{5}$$

It follows that Equation 5 can be expanded over each file in a malware family, and over each family in the set of malware $\mathbb{M}$. This software cutting process allows us to generate very large datasets for model training. As an example, given a malware family with two samples, each 1 megabyte $(1,048,576$ bytes) in length, and a set of benign software, each sample 1 megabyte in length, previous works in this area would be able to generate two triplet pairs [13]–[15]. In contrast, our software cutting method allows for $2,000$ unique triplet pairs. This wide range of possible training examples from a small set of software allows for a more diverse training set given the same malware samples compared to previous work.

IV. MECHA.GEN

Mecha.gen is a gym environment to aid in network generalization. Given an embedding network, a training set, and a support set, Mecha.gen uses an RL technique to further train the embedding network.

A. Gym Environment

In previous works, as well as in evaluating the Mecha.emb system, we found that the embedding network is strong at creating similar embeddings for samples in the same family but fails to make a great distance between samples from different families [13]. To aid with the problem of embedding separation, we train our embedding network in a simulated deployment gym. This gym is Mecha.gen. Mecha.gen is a secondary training for the Mecha.emb embedding network to promote generalization over different families. We will now describe the Mecha.gen gym in detail.

Given an embedding network $f(\cdot)$, a training set, and a support set, we take an EM approach to training the network. EM is a two-step algorithm for estimating underlying parameters to a distribution [26]. In this case, the distribution is the probability distribution that a training sample is within each family used for the support set. Our network, $f(\cdot)$, is an estimator of our probability distribution that is optimized through our EM learning process. We say that the distribution we are trying to estimate is the perfect malware embedder. Given that for both the training and support set we do not have the perfect embeddings, we require an EM approach to iteratively train the network for variant classification. First, the probability distribution of each training sample is found in batches and trained into the network. Second, after each training epoch, the embedding network, $f(\cdot)$, can yield more accurate embeddings, so each byte string in the support set is embedded with the updated parameters of $f(\cdot)$, and the training continues. Such an algorithm is required for training, due to both the training and support set being estimations of an unknown distribution.

The network training process of our EM algorithm is as follows. First, our network embeds all samples in the support set and stores them for training. Second, our network embeds samples in the training set in batches. For each batch of training samples, the Mecha.gen loss is derived and propagated through the embedding network $f(\cdot)$. Once all training batches have been evaluated, the embedding network $f(\cdot)$ embeds the support set with the newly trained parameters. This iteration of training matches that of an EM system.

The Mecha.gen training algorithms are shown in Algorithm 1 and Algorithm 2. Algorithm 1 is the expectation step of the EM method. For the expectation algorithm, the testing and support sets are required along with the embedding function and the chosen optimizer. As discussed above, the embedding function is Mecha.emb, and the optimization algorithm is Adam. Line 1 loops through each batch of data in the training set. Line 2 initializes the loss variable L. Line 3 iterates over each sample b in the batch B. Line 4 initializes the distribution to an empty array. Line 5 iterates over each family set in the support set. Line 6 finds the shortest distance between the testing samples and all the embeddings in family F. That shortest distance is then added to the family distribution. Line 8 normalizes the distribution with an Euclidean-ordered normalization algorithm. Line 9 calculates the reward for sample b by finding the probability that sample b is in the correct family from the family distribution D. Line 10 calculates the

Algorithm 1: Mecha.gen expectation algorithm.

Input: embedding function $f(\cdot)$, optimizer function $a(\cdot,\cdot)$
Output: embedding function $f(\cdot)$
Data: testing set T, support set S

1 **for** B *in* T **do**
2 $L = []$
3 **for** b *in* B **do**
4 $D = []$
5 **for** F *in* S **do**
6 D.add $\left(\min_{s \in F} \|f(b) - s\|_2^2\right)$
7 **end**
8 $D = \text{normalize}(D)$
9 $r = D[b.\text{true\_family}]$
10 $l = 1 - r$
11 L.add(l)
12 **end**
13 f.update_weights$(a(B, L))$
14 **end**
15 **return** $f(\cdot)$

Algorithm 2: Mecha.gen maximization algorithm.

Input: embedding function $f(\cdot)$, family size x
Output: support set S
Data: support super set S'

1 $S = []$
2 **for** F' *in* S' **do**
3 $F = []$
4 $F' = \text{random\_sample}(F', x)$
5 **for** s' *in* F' **do**
6 $s = f(s')$
7 F.add(s)
8 **end**
9 S.add(F)
10 **end**
11 **return** S

loss from the reward. Line 11 adds the sample loss to the batch loss set. Line 13 updates the weights to the embedding function f from the batch of data, the loss, and the optimizer. Line 15 returns the embedding function with the new weights. Algorithm 2 is the maximization step of the EM algorithm. Given the embedding function, the family size, and the support super set, a support set is generated for training. Line 1 initializes the support set S. Line 2 iterates over each family in the support super set. Line 3 initializes the family set F. Line 4 takes of a random sample of size x from family F'. Line 5 iterates over each sample in the family F'. Line 6 embeds the sample. Line 7 adds the embedding to the family set F. Line 9 adds the family set to the support set S. Line 11 returns the updated support set for training in Algorithm 1. It is important to note that the first step of the EM training is initializing the first support set with Algorithm 2.

B. EM Loss

For the Mecha.gen training process, a novel EM loss must be defined. Due to approximating probability distribution being non-differentiable, we take a reinforcement learning approach for creating a reward based on the probability distributions of each training sample. For Mecha.gen training, we require a support set $\mathbb{S} = \{\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_n\}$ of n sets of malware families. Each $\mathbb{F}_i = \{e_{i,1}, e_{i,2}, \ldots, e_{i,n}\}$ is a set of byte strings from malware family i. For a given training sample s in malware family $0 \leq t \leq n$, our reward is the probability that sample s is in family t. To find this probability, we first define the family minimum distance function $m(\cdot,\cdot)$ as 6

$$m(a, \mathbb{E}) = \min_{e \in \mathbb{E}} \|f(a) - f(e)\|_2^2 \tag{6}$$

where a is a single byte string, and $\mathbb{E}$ is a set of malware byte strings. For our sample s in family t we find the family minimum vector $\boldsymbol{m} =$

$$(m(s, \mathbb{F}_1), \ldots, m(s, \mathbb{F}_n)) \tag{7}$$

where the value in index i of vector $\boldsymbol{m}$ is the shortest distance between the training sample s and the embeddings in malware family $\mathbb{F}_i$. We calculate the family probability vector as

$$\boldsymbol{p} = \|\boldsymbol{m}\| \tag{8}$$

this normalizes the family minimum vector so the sum of all elements in $\boldsymbol{p}$ is 1, making each element $\boldsymbol{p}_i$ the probability that sample s is in family i. The reward for training sample s is $r = \boldsymbol{p}_t$. The loss propagated into the network is $1 - r$. As described above, the loss is averaged over batches before being propagated through the network. In experiments, we train in the Mecha.gen training process for 100 epochs using the Adam Stochastic Gradient Descent method [25] with a learning rate of 0.000001. These values were chosen based on empirical hyper-parameter tuning to increase convergence time within 24-hour time frame.

V. Mecha.match

The Mecha.match system is an open set approximate nearest neighbour search algorithm that builds on Hierarchical Navigable Small World (HNSW) done by Malkov and Yashunin in [21]. In deployment, the Mecha.match system has a five-step process for an incoming malware sample.

- Step 1: Given a set of known malware samples, embed all byte strings of length 1,024 and insert them into the network.
- Step 2: For incoming malware, run the sample through the Mecha.emb network to generate an embedding for each 1,024 length byte string in the malware sample.
- Step 3: Find the first nearest neighbour for each embedded byte string.
- Step 4: For each of the nearest neighbours, if the neighbour is outside of the predefined known threshold τ, set the class of the byte string as *'unknown'*.
- Step 5: Set the predicted class of the malware sample by majority vote decision fusion on classes of the embedded byte strings.

This method allows for fast classification of malware, as well as zero-day classification without increasing algorithm order.

The work of [21] proposes an approximate K-nearest neighbour search based on navigable small world graphs with a controllable hierarchy. Navigable small world networks are networks with logarithmic or algorithmic scaling of the greedy graph routine [27], [28]. Steps 1 and 3 of the Mecha.match system expand on [21].

Step 1 of Mecha.match is constructing a graph structure by iteratively inserting embedded malware byte strings. The insertion of a single embedding is as follows. Starting at the highest layer of the graph, a greedy algorithm is used to find the nearest neighbours of the incoming embedding. Then, the insertion algorithm moves to find the nearest neighbours of the next layer down, following the connection from the previously found nearest neighbours. This process continues for all layers in the hierarchy. The greedy algorithm from [21] is built on the algorithm from [29]. Once all levels in the hierarchical graph have been evaluated, the incoming sample is connected to the nearest neighbour found in the iterative search process. In our experiments, the greedy algorithm looks for the single closest neighbour in each level of the hierarchy. As well, following the Euclidean distance loss used for training the Mecha.emb system, we use Euclidean distance for measuring the distance of samples in the graph.

Prior to Step 3, an incoming malware sample is pre-processed in Step 2. This pre-processing uses the Mecha.emb system to embed the malware sample byte strings into the same vector space as the stored malware embeddings. For each byte string in the incoming sample, the byte string is embedded and awaits classification.

Step 3 of the Mecha.match system is classifying each embedding from an incoming malware sample. The search algorithm follows the insertion method. Given an incoming sample, the graph is iteratively searched with a greedy algorithm for the single closest neighbour to the incoming sample.

Step 3 is run for each embedding generated for a single sample. For each embedding, the classification and the distance between the incoming embedding and its nearest neighbour is saved. Step 4 of the Mecha.match system is zero-day classification. If an incoming sample is outside the boundary of known malware embeddings, it is classified as a zero-day sample. For an embedding, if the distance between the embedding and its nearest neighbour is greater than a chosen τ, then the classification of the embedding is set to *'unknown'*.

The final step of the Mecha.match system (Step 5) is the decision fusion of the embedding classifications. Given the classification of all of the embeddings for a malware sample, the most popular classification is set as the predicted classification of the sample.

In a real-world environment, once a sample has been classified, all embeddings are added to the Mecha.match graph with the predicted class as the ground-truth classification. For evaluating the Mecha system, testing samples were not added to the graph after classification. For evaluation, we use a $\tau = 0.01$.

TABLE II
PARAMETERS USED FOR TRAINING THE MECHA SYSTEM.

Parameter	Macha.emb	Mecha.gen
Optimizer	Adam	Adam
Learning Rate	1.00E-04	1.00E-06
Batch size	128	128
Training Epochs	10	100
Margin α	0.2	$\sim$

VI. EXPERIMENTS

We required three different datasets for conducting validity experiments on the Mecha system. Multiple datasets of chronologically categorized data were used. In experiments, we aimed to address the following research questions:

1) Does Mecha recognize existing malware? (RQ1)
2) Does Mecha recognize new (zero-day) malware? (RQ2)
3) How does Mecha perform with a varying number of families? (RQ3)

We conducted five experiments to demonstrate the validity of the Mecha system. The first two experiments were conducted to show that the Mecha system was generalizable to different malware families. These two experiments addressed RQ1. The second two experiments showed if the Mecha system could reliably separate known malware families from unknown malware families. These two experiments addressed RQ2. These experiments observed how Mecha would perform in a real-world scenario where unseen samples are classified by the network. We used the results to infer the performance of Mecha in a real-time scenario and further show its practicability. These experiments do not analyze the samples further after Mecha classification. In a real-world scenario, the decision on further analysis would be chosen by the CTI professionals in SoC or other threat ingestion centers, such as extended sandboxing or bare-metal evaluation. The final experiment observed how Mecha performed with different family set sizes. This experiment addressed RQ3. As well, multiple state-of-the-art ML methods and Siamese-based malware family detection methods were compared against the Mecha system in family matching ability. The Mecha system was developed on a Linux machine with 16 cores Xeon Gold 2.3/3.9 GHz, a single RTX6000 with 24GB of VRAM, and 200GB of memory. Mecha was developed in Python using open-source packages such as TensorFlow, NumPy, and Pandas. The parameters used for training the Mecha system can be seen in Table II. The loss for training the Mecha.emb network can be seen in Fig. 4. The final training loss was 0.0028 and final validation loss was 0.1777. The loss for training the Mecha.emb network in the Mecha.gen deployment simulation can be seen in Fig. 5. The final training loss was 0.0639 and final validation loss was 0.0647.

A. Dataset

Three different datasets in total were used for training and evaluating the Mecha system: *training-2021*, *testing-2022-q1*, and *testing-2022-q2*. The training dataset, *training-2021*, was a collection of real world malware samples identified in the year

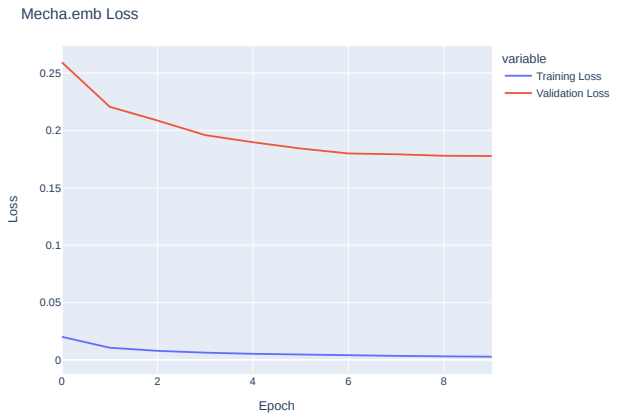

Fig. 4. Training and validation loss for the Mecha.emb network in triplet training. Final training loss of 0.0028 and final validation loss of 0.1777.

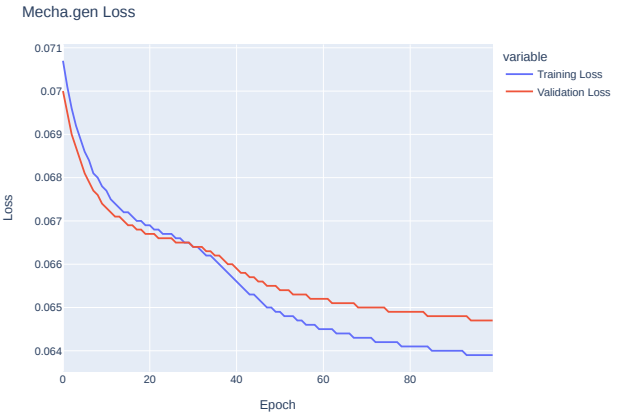

Fig. 5. Training and validation loss for the Mecha.emb network in Mecha.gen deployment simulation training. Final training loss of 0.0639 and final validation loss of 0.0647.

2021, and benign samples from varying years. The two testing sets were datasets of malware identified in the first and second quarters of 2022. We separated our data chronologically to better simulate a real-world malware triage environment. We will now describe each dataset in detail. A visualization of the datasets can be seen in Fig. 6.

All malware samples in the Mecha training corpus were collected from the online repository MalwareBazaar [30]. MalwareBazaar is a popular open malware project that provides malware samples to the CTI community [30]. The labelling approach for temporal information on malware samples was the date the sample was first uploaded to the database. MalwareBazaar conducted its own family classification of malware samples with VirusTotal; we used their sample classifications as the ground-truth for our experiments. All malware samples used had unique hashes associated with the variant. Benign files used were collected from various online repositories.

The Mecha training dataset, *training-2021*, was a corpus of malware and benignware used for generating the triplet pairs for Mecha.emb training, and the testing and support sets used for Mecha.gen training. The Mecha training corpus was comprised of 50,248 malware samples sampled from 175 families, and 7,828 benign samples from varying software vendors. All samples used were Windows Portable Executable files. As described above, the benign samples were only used for generating the *negative* samples of the triplet pairs. 1,000,000 triplet pairs were generated from the corpus for the Mecha.emb training. Triplet pairs were randomly sampled byte strings from the corpus of malware and benignware with replacement. As described above, Mecha.gen training process required two datasets for a single training cycle: support set and embedding set. The support set acted as a simulated malware repository that a CTI organization would use to compare incoming samples against; it consisted of malware samples embedded at the beginning of the training cycle. The embedding set consisted of malware samples that were embedded throughout the training cycle and were classified based on the support set. Both the support set and the embedding set are sampled randomly from the training set at the beginning of each training epoch. This ensures no leakage occurs from the evaluation dataset, as both the support set and embedding set are exclusively sampled from the evaluation dataset later during performance evaluation. This separation maintains the integrity of the results and prevents unintended influence between the training and evaluation phases. Two samples were chosen for each of the 175 families in the training data, and all 1,024 length byte strings of those samples comprised the support set. As well, the testing set was the 1,024 byte strings of two samples randomly sampled from each family in the training data. All samples were randomly chosen uniformly to best represent the corpus of malware without adding any selection bias. Two datasets were required for testing the Mecha system. These sets where *testing-2022-q1*, and *testing-2022-q2*. These datasets were malware samples identified in the first and second quarter of 2022 respectively. As previously stated, all of the malware used for training and evaluating the efficacy of the Mecha system was separated to better simulate a real-world environment. Two separate testing sets were required due to the nature of the Mecha system. The Mecha system required a support set of samples that were stored for matching. We used the testing set from the first quarter of the year as the support set, and we tested the matching ability of the Mecha system on the second quarter dataset. These two datasets were further separated between in-sample and out-of-same families from the data. As stated, the training dataset, *training-2021*, contained 175 families of malware. The in-sample testing datasets, testing-2022-q1-in, and testing-2022-q2-in contained samples from 25 families that were sampled from the 175 training families. The out-of-sample testing datasets, testing-2022-q1-out, and testing-2022-q2-out contained malware samples from 25 families that were not in *training-2021*. Please refer to Table III for the list of families, as well as a number of samples per family used in the *testing-2022-q1*, and *testing-2022-q2* sets. It is important to note that there is no overlap in distinct malware samples between all of the discussed datasets.

TABLE III
LIST OF FAMILIES USED IN THE IN-SAMPLE AND OUT-OF-SAMPLE SETS. THE NOTATION IS : FAMILY NAME (NUMBER OF SAMPLES IN Q1 SET, NUMBER OF SAMPLES IN Q2 SET).

In-sample Families - Total (6278, 2955)					
agenttesla (3679, 1658)	snakekeylogger (1097, 600)	avemariarat (413, 227)	nanocore (314, 157)	njrat (312, 110)	gozi (86, 38)
coinminer (86, 18)	yellowcockatoo (39, 30)	coinminer.xmrig (37, 2)	tofsee (28, 12)	zeus (19, 7)	danabot (19, 15)
trickbot (18, 17)	urelas (17, 4)	a310logger (17, 12)	virlock (12, 11)	matanbuchus (8, 7)	runningrat (8, 3)
ircbot (7, 6)	cryptbot (7, 4)	sodinokibi (7, 7)	chaos (7, 3)	dridex (6, 3)	blackshades (6, 2)
kutaki (2, 2)					
Out-of-sample Families - Total (756, 504)					
smoke loader (310, 120)	resur (114, 114)	emotet (50, 50)	triusor (47, 47)	evora (36, 36)	emotet_b (28, 28)
lockbit (19, 2)	parite (18, 18)	babdeda (17, 6)	blackguard (11, 2)	bitter (11, 11)	ursnif (11, 11)
vovabol (11, 11)	ketrican (9, 9)	dtrack (9, 9)	shifu (7, 4)	mydoom (6, 4)	qqpass (5, 5)
babuk (3, 3)	berbew (3, 3)	fathula (3, 3)	blister (2, 2)	vobfus (2, 2)	thanos (2, 2)
blackout (2, 2)					

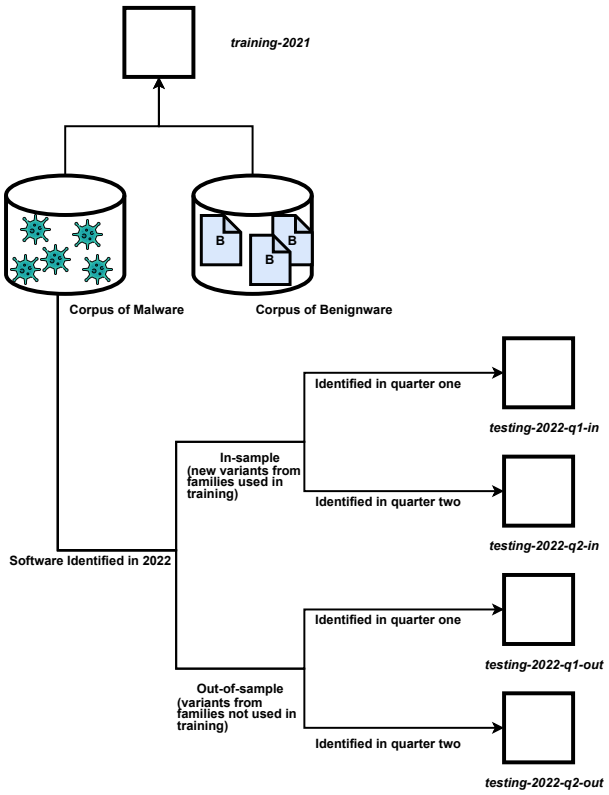

Fig. 6. A visualization of the datasets required for training and testing the Mecha system. The set *training-2021* was used for training the Mecha system. The quarter one sets were used as support sets in the experiments, and the quarter two sets were used as testing sets in the experiments.

B. Benchmarks

The Mecha system was compared against 14 other methods for the family matching experiments. Three of the methods were the Malkov and Yashunin approximate nearest neighbour algorithm, [21], with different input data. The input data tested with [21] was raw malware byte string, L2 normalized malware byte string, and average pooled malware byte string. These benchmarks were chosen as a method of comparing the entire Mecha system against different subcomponents of the Mecha system used together. As well, we compared Mecha against two models similar to the Mecha network with varying input dimensions. The input dimensions chosen were 512 and 256. These models were referred to as Mecha-512 and Mecha-256, respectively. These models were compared against Mecha to evaluate the performance difference of using varying input sizes. Five popular ML classifiers were also implemented to compare against the Mecha system. The ML methods compared against the Mecha system were Quadratic Discriminant Analysis (QDA) [31], Ada Boost [32], Decision Tree [33], Random Forest [34], and Gaussian Naive Bayes (NB). We compared these methods to ensure the Mecha system had a higher performance than conventional state-of-the-art ML-based classifiers. It is important to note that these five benchmarks were not originally developed for malware family classification but have shown great success in many classification tasks. Unlike these methods, the Mecha system has been optimized specifically for the task of malware family similarity analysis. As well, we compare the Mecha system against the similarity hashing method ssdeep [16]. The ssdeep method is used in many industry malware detection systems, and we compared Mecha against ssdeep to further observe real-world viability. The ssdeep solution performed a pairwise comparison between the incoming sample and the support set. The results for the pairwise comparison were used to build a distribution for the family classification of the malware sample based on the highest similarity score between the incoming sample and each family. The Siamese-based malware similarity networks proposed by Hsiao *et al*. and Conti *et al*. were implemented to compare classification ability in Siamese networks depending on input data, and output dimensional. These benchmarks especially compared the ability of the embedding network compared to Mecha. As well, these benchmarks used a single modality input similar to Mecha. It is important to note that these two networks have an input of two sample images and output a single similarity score [14], [15], this is in contrast to the single sample input of the deployed Mecha system. Due to this, the comparison between a single testing sample, and the entire support set required significantly more time than all other methods used in the experiments. Finally, the Mecha.emb, and Mecha.match systems were compared to the full Mecha system as an ablation test to determine the effect of the Mecha.gen EM training on the classification ability of the Mecha system. All matching algorithms were given a 24-hour time limit for inference, after that, all results

TABLE IV
RESULTS ON THE MATCHING OF *testing-2022-q2-in* ON *testing-2022-q1-in*. THE METRICS REPORTED ON THIS TABLE DESCRIBE THE PERFORMANCE OF DIFFERENT MODELS ON THE IN-SAMPLE TESTING DATA. GIVEN A SUPPORT SET OF MALWARE SAMPLES (*testing-2022-q1-in*) FROM FAMILIES WITHIN THE TRAINING DATA, THE MODELS WERE TASKED WITH PREDICTING THE FAMILIES OF THE TESTING SET (*testing-2022-q1-in*).

Baseline	AUC	Accuracy	F1	Precision	Recall	Optimal Threashold	Mean Embedding Distance	Sample Inference Time (seconds)
QDA [31]	0.8412	0.8133	0.2719	0.1611	0.8714	0.0169	∼	1.2825
Ada Boost [32]	0.7681	0.9586	0.5199	0.4844	0.5611	0.1111	∼	0.1087
Decision Tree [33]	0.7914	0.9619	0.5601	0.5206	0.6061	0.0011	∼	0.0276
Random Forest [34]	0.7714	0.9649	0.5611	0.5611	0.5611	1.0000	∼	0.0327
Gaussian NB	0.6233	0.6205	0.1166	0.0643	0.6264	0.0017	∼	0.0823
ssdeep [16]	0.9861	0.9820	0.8149	0.6921	0.9905	0.2494	∼	0.0127
HNSW [21]	0.9016	0.8362	0.3221	0.1930	0.9726	0.0225	4107985.532	0.5016
L2 + HNSW [21]	0.9479	0.9246	0.5082	0.3439	0.9733	0.0737	0.2471	0.6679
Average Pooling + L2 + HNSW [21]	0.9794	0.9741	0.7528	0.6091	0.9851	0.1116	0.0361	0.2078
Hsiao *et al.* [14]	0.5072	0.2051	0.2752	0.1609	0.9500	0.4981	∼	10.4836
Conti *et al.* [15]	0.5000	0.6255	0.0000	0.0000	0.0000	2.0000	∼	13.0884
Mecha.emb + Mecha.match	0.9529	0.9436	0.5773	0.4122	0.9631	0.1055	0.3375	0.5096
Mecha-256	0.8409	0.8813	0.3495	0.2238	0.7970	0.0154	0.7801	0.3233
Mecha-512	0.9914	0.9841	0.8337	0.7152	0.9993	0.1430	0.0067	0.4636
Mecha	0.9982	0.9984	0.9804	0.9634	0.9980	0.3185	0.0040	0.6150

TABLE V
RESULTS ON THE MATCHING OF *testing-2022-q2-out* ON *testing-2022-q1-out*. THE METRICS REPORTED ON THIS TABLE DESCRIBE THE PERFORMANCE OF DIFFERENT MODELS ON THE OUT-OF-SAMPLE TESTING DATA. GIVEN A SUPPORT SET OF MALWARE SAMPLES (*testing-2022-q1-out*) FROM FAMILIES OUTSIDE THE TRAINING DATA, THE MODELS WERE TASKED WITH PREDICTING THE FAMILIES OF THE TESTING SET (*testing-2022-q1-out*).

Baseline	AUC	Accuracy	F1	Precision	Recall	Optimal Threashold	Mean Embedding Distance	Sample Inference Time (seconds)
QDA [31]	0.9894	0.9924	0.9119	0.8481	0.9861	0.2719	∼	0.7909
Ada Boost [32]	0.7136	0.8298	0.2163	0.1326	0.5873	0.0019	∼	0.0423
Decision Tree [33]	0.6532	0.9329	0.2941	0.254	0.3492	0.0154	∼	0.0100
Random Forest [34]	0.6377	0.9415	0.2961	0.2855	0.3075	0.2127	∼	0.1074
Gaussian NB	0.6262	0.5452	0.1116	0.0605	0.7143	0.0116	∼	0.0397
ssdeep [16]	0.9993	0.9987	0.9834	0.9674	1.0000	0.4831	∼	0.0015
HNSW [21]	0.9748	0.9552	0.6403	0.4718	0.996	0.0667	4520590.163	0.1389
L2 + HNSW [21]	0.9209	0.8829	0.3966	0.2497	0.9623	0.0222	0.3297	0.0431
Average Pooling + L2 + HNSW [21]	0.9669	0.9638	0.6820	0.5258	0.9702	0.1760	0.0741	0.0197
Hsiao *et al.* [14]	0.5072	0.2051	0.2752	0.1609	0.9500	0.4981	∼	10.4836
Conti *et al.* [15]	0.5000	0.8412	0.0000	0.0000	0.0000	2.0000	∼	1.8012
Mecha.emb + Mecha.match	0.9520	0.9260	0.5146	0.3489	0.9802	0.0625	0.3538	0.0615
Mecha-256	0.6145	0.6705	0.1185	0.0663	0.5536	0.0166	0.7723	0.0720
Mecha-512	0.9974	0.9986	0.9824	0.9691	0.9960	0.3984	0.0407	0.0668
Mecha	0.9980	0.9962	0.9545	0.9130	1.0000	0.3485	0.0073	0.0822

are predicted as 0. For comparing the different methods in our experiments, we used the metric Area under the ROC curve (AUC), accuracy, precision, recall, and F1-Score. As well, all metrics were calculated with the optimal threshold determined by maximizing the Youden's J statistic of the ROC curve.

C. RQ1

The first two experiments we conducted on the Mecha system validated the generalizability of the Mecha system. We performed a simulated real-world malware triage environment using the *testing-2022-q1* dataset as stored data and the *testing-2022-q2* dataset as new incoming samples. We performed this experiment twice, first with the in-sample data, and then with the out-of-sample data. Along with verifying that the Mecha system can reliably match unseen malware to the correct family, this experiment verified that the Mecha training process did not overfit the Mecha.emb network to the training families. These experiments are referred to as in-sample generalizability and out-of-sample generalizability.

The results for the two generalization experiments can be seen in Table IV and Table V. As can be seen from Table IV, Mecha performed exceptionally well at classifying the trained malware families, with an AUC of 0.9982 and accuracy of 0.9984 on the in-sample set. This clearly showed Mecha's

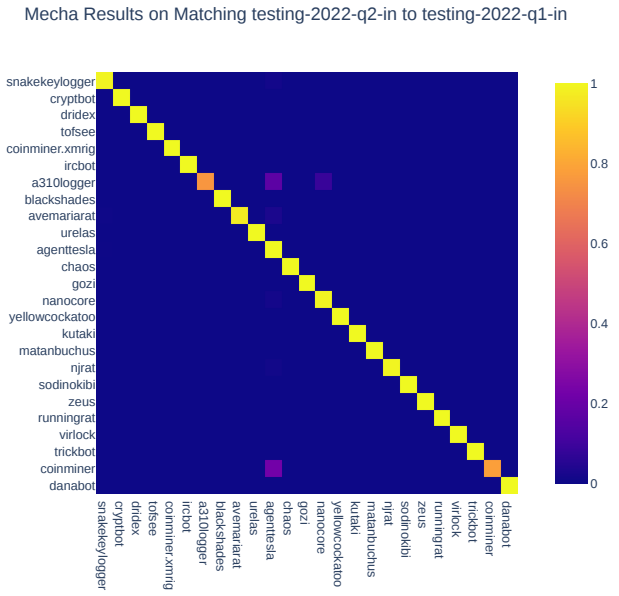

Fig. 7. Family matching by Mecha on the in-sample data (samples that are variants in training families, but were not used in training). These results correspond to the performance of the Mecha system shown in Table IV.

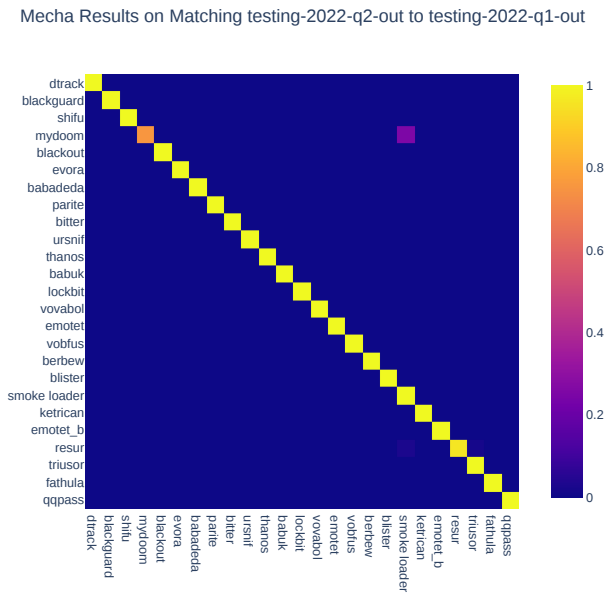

Fig. 8. Family matching by Mecha on the out-of-sample (samples that are variants of families not used in network training) data. These results correspond to the performance of the Mecha system shown in Table V.

ability to accurately match malware variants to the correct family. As well, it can be seen that the ablation test showed performance enhancement from training in the Mecha.gen generalization RL environment. For F1, precision, and recall, the Mecha system performed very well with high results in all metrics. This further showed the balance of Mecha when matching correct samples as well as indicating when two samples are not from the same family. For sample inference time, the fastest model was the ssdeep baseline with a sample inference time of 0.0127 seconds compared to a sample inference time of 0.6150 seconds for Mecha. It can also be seen that the embeddings from the Mecha system had a significantly lower mean embedding distance than other distance-based evaluation tools, further demonstrating the embedding power of the Mecha.emb and Mecha.gen training. Compared to Mecha-256 and Mecha-512, Mecha performed the best with an AUC of 0.9980. It is interesting to note performance did not decrease significantly between Mecha and Mecha-512 (AUC of 0.9914), but the performance of Mecha-256 (AUC of 0.8409) was very poor compared to the other two Mecha models. Further, it can be seen that the ssdeep baseline performed very well against the in-sample set with an AUC of 0.9861 and an accuracy of 0.9820. These results were very similar to the Mecha results. The similarity in results further showed the real-world viability of the Mecha system. The classification of each malware family by the Mecha system on the in-sample data can be seen in Fig. 7. As can be seen from Fig. 7, the only families with a significant mismatch of samples were the a310logger family and the coinminer family. For the out-of-sample data, Mecha performed very well with an AUC of 0.9980 and accuracy of 0.9962. These results were very similar to the ssdeep solution which had an AUC of 0.9993 and an accuracy of 0.9987. Further, the results of Mecha were similar to the results of Mecha-512, which had an AUC of 0.9974 and an accuracy of 0.9986. It is interesting to note that the accuracy of Mecha-512 was higher than Mecha. Also, similar to in-sample testing, the Mecha-256 system performed very poorly with an AUC of 0.6145 and accuracy of 0.6705. The two generalization experiments showed that using an input size of 512 did not affect model performance significantly, but using an input size of 256 dramatically decreased model performance. Similar to the in-sample test, the Mecha system had the shortest mean embedding distance. It is important to note that the Mecha system did not have the lowest sample inference time. The solution with the lowest sample inference time was the ssdeep method with an inference time of 0.0015 seconds per sample. The matching results on the out-of-sample data can be seen in Fig. 8. From the heatmap, it can be seen that only the MyDoom family was significantly misclassified with a match accuracy of 0.75. Referring to RQ1, these results clearly show that the Mecha system was able to identify malware samples that are known to the support set.

D. RQ2

The second two experiments validated the zero-day detection ability of the Mecha system. The third experiment validated zero-day family detection as a binary classification problem. Given a sample, the Mecha system was tasked with classifying the malware as either known or unknown. The fourth experiment validated the zero-day family detection in a full malware triage simulation environment. Given an incoming sample, Mecha was tasked with categorizing the sample into a known malware family or categorizing the sample as unknown. For these experiments, the *testing-2022-q1-in* dataset was the simulated stored data, and both the *testing-2022-q2-in* and *testing-2022-q2-out* datasets were the new incoming samples.

Table VI shows the results for the zero-day detection as a binary classification problem experiment. As can be seen, the Mecha system outperformed all baselines significantly in this experiment. As well, the Mecha system had an AUC of 0.9962 and an accuracy of 0.9977 making the model almost perfect at detecting if an incoming sample was known to the system or not. Also, the F1, precision and recall are all almost 1 with the lowest of the three being precision with a score of 0.9901. This is in contrast to the ssdeep benchmark, which had an AUC of 0.5 and an accuracy of 0.8543. This low AUC was caused by the lack of an unknown mechanism in the ssdeep benchmark. Compared to the varying input dimension benchmarks, Mecha had a slightly better performance. Where Mecha-256 and Mecha-512 had AUC of 0.9745 and 0.9723, respectively, Mecha had an AUC of 0.9962. Similar to previous experiments, the sample inference time for Mecha was half a second on average.

Table VII shows the results of the zero-day family matching experiment. The Mecha system showed the best performance with an AUC of 0.9978 and accuracy of 0.9979. As well, the F1 precision, and recall, were all above 0.95, with the lowest metric being precision with a score of 0.9512. These results were similar to, but better than the results of Mecha-256 and Mecha-512. Mecha-512 had an AUC of 0.9849 and

TABLE VI
RESULTS FROM THE ZERO-DAY PREDICTION EXPERIMENT. THE METRICS ON THIS TABLE DESCRIBE THE PERFORMANCE OF THE MODELS AT THE TASK OF CLASSIFYING IF AN INCOMING SAMPLE IS FROM A ZERO-DAY FAMILY. GIVEN A SUPPORT SET OF MALWARE SAMPLES (*testing-2022-q1-in*), THE MODELS ARE TASKED TO PREDICT IF INCOMING SAMPLES ARE FROM KNOWN OR UNKNOWN FAMILIES IN THE *testing-2022-q2-in* AND *testing-2022-q2-out* SETS. IF THE INCOMING SAMPLE IS OUTSIDE OF A GIVEN THREASHOLD (τ) TO ITS CLOSEST NEIGHBOUR, IT IS PREDICTED AS UNKNOWN.

Baseline	AUC	Accuracy	F1	Precision	Recall	Optimal Threashold	Mean Embedding Distance	Sample Inference Time (seconds)
QDA [31]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	1.2087
Ada Boost [32]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	0.0806
Decision Tree [33]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	0.0231
Random Forest [34]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	0.0309
Gaussian NB	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	0.0721
ssdeep [16]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	∼	0.0117
HNSW [21]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	4265060.999	0.4478
L2 + HNSW [21]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	0.2619	0.4382
Average Pooling + L2 + HNSW [21]	0.5000	0.8543	0.0000	0.0000	0.0000	1.0000	0.0509	0.1834
Hsiao *et al.* [14]	0.5000	0.0000	0.0000	0.0000	0.0000	1.0000	∼	1.4371
Conti *et al.* [15]	0.5000	0.0000	0.0000	0.0000	0.0000	1.0000	∼	1.8335
Mecha.emb + Mecha.match	0.6890	0.8103	0.4431	0.3872	0.5179	0.9548	0.3627	0.4534
Mecha-256	0.9745	0.9720	0.9104	0.8515	0.9782	0.0960	0.0076	0.5453
Mecha-512	0.9723	0.9737	0.9149	0.8655	0.9702	0.3156	0.0197	0.4121
Mecha	0.9962	0.9977	0.9921	0.9901	0.9940	0.3915	0.0064	0.5031

TABLE VII
RESULTS FROM THE FAMILY CLASSIFICATION WITH ZERO-DAY SAMPLES EXPERIMENT. THE METRICS ON THIS TABLE DESCRIBE THE PERFORMANCE OF THE MODELS AT THE TASK OF CLASSIFYING ALL INCOMING SAMPLES TO A SPECIFIC FAMILY, OR AS UNKNOWN. GIVEN A SUPPORT SET OF MALWARE SAMPLES (*testing-2022-q1-in*), THE MODELS ARE TASKED TO CATEGORIZE THE INCOMING SAMPLES IN THE *testing-2022-q2-in* AND *testing-2022-q2-out* SETS. IF THE INCOMING SAMPLE IS OUTSIDE OF A GIVEN THREASHOLD (τ) TO ITS CLOSEST NEIGHBOUR, IT IS CATEGORIZED AS UNKNOWN.

Baseline	AUC	Accuracy	F1	Precision	Recall	Optimal Threashold	Mean Embedding Distance	Sample Inference Time (seconds)
QDA [31]	0.8332	0.7977	0.2274	0.1308	0.8714	0.0169	∼	1.2087
Ada Boost [32]	0.7659	0.9566	0.4692	0.4032	0.5611	0.1111	∼	0.0806
Decision Tree [33]	0.7895	0.9603	0.5105	0.4410	0.6061	0.0011	∼	0.0231
Random Forest [34]	0.7698	0.9642	0.5170	0.4793	0.5611	1.0000	∼	0.0309
Gaussian NB	0.6114	0.5684	0.0943	0.0508	0.6575	0.0013	∼	0.0721
ssdeep [16]	0.9872	0.9840	0.8090	0.6837	0.9905	0.2494	∼	0.0127
HNSW [21]	0.8990	0.8383	0.2895	0.1703	0.9641	0.0251	4265060.999	0.4478
L2 + HNSW [21]	0.9379	0.8973	0.3952	0.2474	0.9814	0.0588	0.2619	0.4382
Average Pooling + L2 + HNSW [21]	0.9719	0.9637	0.6485	0.4845	0.9807	0.1060	0.0509	0.1834
Hsiao *et al.* [14]	0.5000	0.6801	0.0000	0.0000	0.0000	2.0000	∼	1.4371
Conti *et al.* [15]	0.5000	0.9769	0.0000	0.0000	0.0000	2.0000	∼	1.8335
Mecha.emb + Mecha.match	0.9669	0.9423	0.5699	0.3995	0.9936	0.0051	0.3627	0.4534
Mecha-256	0.9923	0.9940	0.9272	0.8715	0.9905	0.1980	0.0076	0.5453
Mecha-512	0.9849	0.9800	0.7917	0.6595	0.9902	0.2104	0.0197	0.4121
Mecha	0.9978	0.9979	0.9739	0.9512	0.9977	0.3056	0.0064	0.5031

an accuracy of 0.9800. Mecha 256 had an AUC of 0.9923 and an accuracy of 0.9940. These results further showed that 1024 is the optimal input dimension of the three tested. The model that performed the third best was ssdeep with an AUC of 0.9872 and accuracy of 0.9840. This was a better result than the ablation Mecha.emb + Mecha.match, but still not as accurate as the entire Mecha system. It is important to note the large difference in performance between most of the models in the binary classification and the family matching experiment. For the family matching, many models previously showed an ability to classify known families, which is causing high success in the zero-day family matching experiment. It shows in the binary classification experiment that almost all of the models have little to no ability to detect if a sample is known or unknown to the model. The classification of the families in the zero-day experiment can be seen in Fig. 9. As can be seen, almost all of the unknown samples (0.9780) were correctly classified as unknown. The other unknown samples were classified as the Danabot family.

E. RQ3

The fifth experiment we conducted on the Mecha system was to observe how Mecha performed with varying sizes of malware families. The purpose of this experiment was to determine whether family size affected Mecha's performance. In this experiment, we recreated the first experiment of in-sample family matching with a varying number of malware families. Originally, there were 25 families in the in-family set. Subsets of sizes 20, 15, and 10 were used to observe the performance of Mecha with varying family set sizes. It is important to note that larger subsets contain all the families used in the smaller subsets. The subset families were sampled randomly without replacement.

TABLE VIII
RESULTS OF MECHA ON THE MATCHING OF *testing-2022-q2-in* ON *testing-2022-q1-in* WITH A VARYING SUBSET OF FAMILIES.

Total Families	AUC	Accuracy	F1	Precision	Recall
25	0.9982	0.9984	0.9804	0.9634	0.9980
20	0.9993	0.9991	0.9911	0.9827	0.9996
15	0.9994	0.9988	0.9913	0.9827	1.0000
10	1.0000	1.0000	1.0000	1.0000	1.0000

The results for the family size experiment can be seen in Table VIII. As can be seen, the size of the family appeared to be directly related to the performance of Mecha. As the

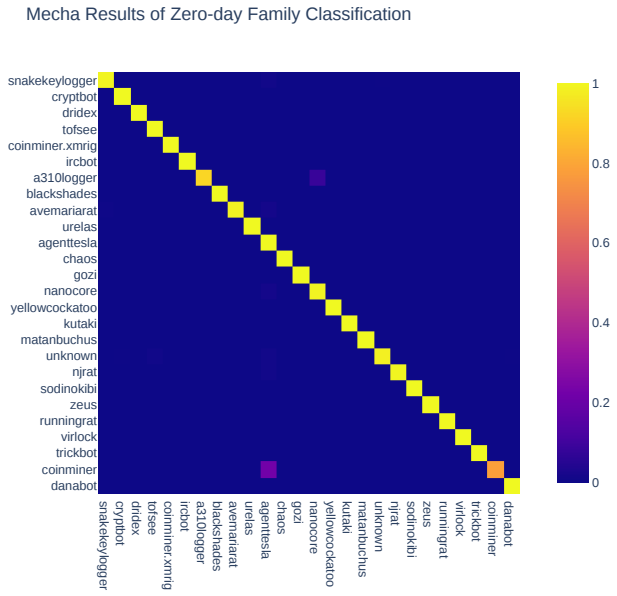

Fig. 9. Family matching by Mecha in zero-day family matching experiment. These results correspond to the performance of the Mecha system shown in Table VII.

total number of families decreased toward 0, the performance metrics increased to 1. It is interesting to note that the accuracy of Mecha for classifying malware into 20 families (0.9991) was higher than the accuracy of Mecha classifying malware into 15 families (0.9988).

F. Further Evaluation

For further evaluation on the ablation study between Mecha.emb + Mecha.match compared to the entire Mecha system, the ROC curves for the two models in the four experiments can be seen in Fig. 10. As can be seen from the graphs, the Mecha system is much closer to an ideal curve than the ablation model. This further validates the aid in model training the reinforcement learning environment provides to the Mecha system.

Outside of the solutions proposed by Hsiao *et al.* and Conti *et al.* all solutions completed inference time for each experiment within the 24-hour limit. It can be observed that the Mecha system did not perform the fastest in experiments for sample inference. Although some solutions, such as ssdeep [16], Ada Boost [32], Decision Tree [33], Random Forest [34], and Gaussian NB perform sample inference significantly faster than Mecha, there is a decrease in solution performance. Further, for the ssdeep solution there exists the problem of scalability. The ssdeep solution required a linear search of the support set for each sample in the testing set. This linear search had a complexity of $O(n)$ where n is the size of the support set. Whereas the Mecha solution used an approximate nearest neighbour search for each sample in the testing set, which had a search complexity of $O(\log(n))$ where n is the size of the support set. Although the sample inference time was faster in our experiments, at a larger scale ssdeep inference time will increase dramatically compared to Mecha. It is important to note that due to the complexity of the Mecha system, training time was significatly longer than other solutions. Training time was not restricted in our experimentation.

VII. Related Works

Due to the constantly evolving landscape of attack methods, and the vast amount of new malware each day, it is impossible for humans to detect and categorize malware on their own. With recent advancements in artificial intelligence (AI) and ML, learned algorithms have been proposed for solving problems in CTI [13], [19], [24], [35]–[39]. The features used for malware detection in ML can be separated into the categories of static analysis and dynamic analysis [40]. Dynamic analysis involves running a software sample in a sandbox environment and examining the behaviour, whereas static analysis aims to determine if a sample is malicious or benign based on analyzing the code or structure [40]. A popular feature for static malware detection is raw byte sequences [13], [24], [35]. Raw byte sequences have been used in previous works with convolution neural networks for the task of malware detection [13], [24], [35]. To expand from malware detection, static analysis has also shown success in malware similarity analysis using DL models. A popular DL method that has shown recent success is the Siamese Neural Network.

The Siamese Neural Network Architecture is a method of learning for similarity analysis. Siamese networks learn through either a twin or triplet learning algorithm [23], [41], [42]. The Siamese network was first proposed in the work by Bromley *et al.* where the twin Siamese network was proposed for measuring the similarity between human signatures written on a pen-input tablet [42]. The work of measuring the similarity between two images was applied to a more broad image similarity problem by Koch *et al.* [41]. In their work, Koch *et al.* showed a twin Siamese network could be used for comparing images in similarity from categories that only contain a single sample in the training data (One-shot classification). The work of Koch *et al.* has led to the application of twin Siamese networks in many domains, including malware similarity analysis [14], [15]. The twin Siamese network was expanded to the triplet in [23] with FaceNet. FaceNet is a convolution neural network trained to generate embeddings for face images that could be compared by L2-distance [23].

Similar to recent works which use Siamese-based architectures for software similarity analysis [7], [8], [13]–[15], [22], we use static features extracted from the code base of the malware sample. Rather than using image representations of malware executables [7], [8], [14], [15], [43], or features extracted from reverse engineering techniques [13], [22], we use raw byte sequences directly from malware samples to generate embedding representations.

The Siamese network architecture has been used for classifying malware into a discrete set of malware families [7], [8]. These networks are fitted with a Softmax activation head. The Softmax activation head is used to classify the two samples into the discrete set of malware families that are in the

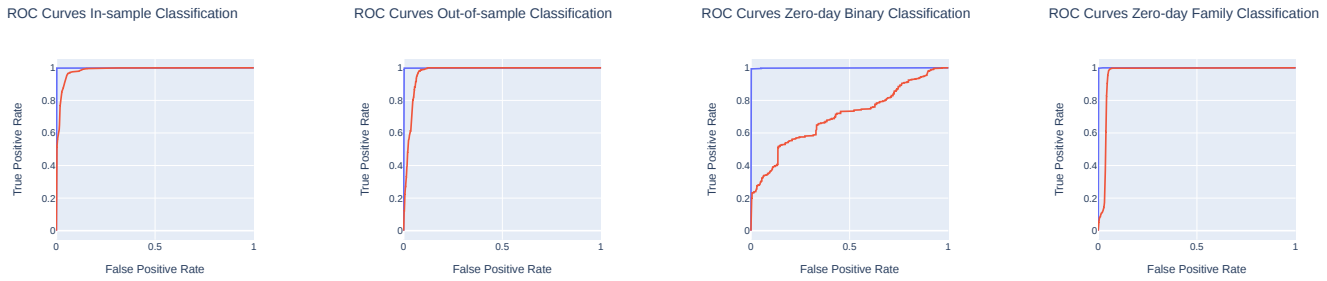

Fig. 10. ROC curves from the four experiments. In all curves, the Mecha system is blue, and the Mecha.emb + Mecha.match model is in red. The optimal ROC curve would have an area of 1.0 under it.

training data. Zhu *et al*. proposed a twin Siamese network with a Softmax activation head for ransomware detection and classification. This method showed success at the few-shot classification of ransomware families [7]. Chen *et al*. furthered the work of Softmax-activated networks by designing a multi-head network that would output a family classification, as well as an embedding for similarity analysis [8].

The works of [14] and [15] used a Twin Siamese architecture for malware similarity scoring. Given an unknown sample and a support set, a similarity score is found for each support sample with the unknown sample. The unknown sample is classified into the same family as the support sample with the highest similarity. Siamese architectures have been shown to work well in few-shot and one-shot learning paradigms [14]. Where conventional ML requires multiple samples in each class, few-shot learning is the method of using a few samples for each category in a classification task. Following this, one-shot learning uses a single sample in a class for network training. Few-shot and one-shot learning have been shown to work well in malware similarity analysis and family classification [14], [44]. Hsiao *et al*. proposed a system for finding the similarity between two malware samples [14]. Using byte-to-pixel images of malware samples as input, Hsiao *et al*. showed the deep framework proposed in [41] could be applied to the problem of malware similarity scoring. Conti *et al*. proposed a multi-network system that would find the similarity score on two malware samples based on a three-channel image (Gray-level matrix image + Entropy graph image + Markov image) [15]. Following Few-shot and One-shot training, the Mecha system has been shown to work well in classifying samples not used in model training. This shows that Mecha is successful at Zero-shot testing.

The work of [13] explored the method of generating embeddings for storage and similarity analysis like the work of [23]. In their work, Molloy *et al.* proposed a Generative Adversarial Network for malware embedding generation with reconstruction [13]. Like the work of [15], Molloy *et al.* used multiple static feature vectors from a single malware sample as input to their network. Instead of using different image modalities, Molloy *et al.* used five features extracted from each sample through static analysis (byte code + import text + string text + byte image + byte image signature) [13].

A conventional method for comparing software samples is through similarity hashing [17], [18], [45]. Similarity hashing is the method of hashing rolling blocks of a sample and comparing blocks through an edit distance to find a similarity score [16]. A popular similarity hashing tool proposed by Kornblum is ssdeep [16]. Similarity hashing tools such as ssdeep have been shown to aid in malware family classification [17], [18].

Similarity hashing tools have been shown to aid ML-based malware analysis systems. Edir [17] proposed a three-step process for malware classification, which involved similarity hashing and supervised learning for malware detection. The approach aimed to reduce the input size for machine learning classifiers by converting the full binary sequence into a single hash, thus avoiding the high computational cost of processing the entire binary at once. However, our proposed learning method is designed to directly process the full binary sequence without overburdening the machine learning model, as it is optimized through a two-stage process. Additionally, our goal focused on identifying similar samples at scale, whereas Edir's [17] objective was to determine which part of the similarity hash contributes more significantly to the detection of maliciousness. Botacin *et al*. have also proposed a solution to malware detection and classification that utilizes similarity hashing and ML [18]. In their work, Botacin *et al*. propose a method of clustering similarity hashes and classifying software through centroid comparison. DBSCAN was used to generate cluster centroids to decrease classification time by comparing similarity to cluster centroids only [18]. Our work differs from the work of Botacin *et al*. Because the Mecha system has been designed for similarity analysis only, as well, our work allows for open-set classification, whereas the work of Botacin *et al*. used a set of cluster centroids for classification. Incremental updates to DBSCAN clusters are challenging in practice due to ongoing concept drift in the underlying data. In contrast, approximate nearest neighbour search methods for embedding space offer finer search granularity in cases of concept shift and serve as more robust update methods.

VIII. Conclusion

In this work, we proposed Mecha, the first multi-ML system for malware variant matching with zero-day family detection. We utilized Siamese learning, as well as reinforcement learning for developing an embedding network that outperforms

the state-of-the-art. As well, we built on HNSW to create an open set approximate nearest neighbour system for embedding family matching. We showed through multiple experiments that the Mecha system is generalized to in-sample and out-of-sample malware families, and can accurately categorize samples into known families with zero-day detection functionality. A limitation of the Mecha system is that it has only been designed for Windows PE malware. With the growing interest in Internet of Things (IoT) infrastructure, future work includes testing the proposed methods against IoT malware [46]. Zero-day attack detection has already been shown to be successful against IoT malware [47], and future work includes building on the Mecha system to combat IoT malware.

REFERENCES

[1] J. A. Marpaung, M. Sain, and H.-J. Lee, "Survey on malware evasion techniques: State of the art and challenges," in *2012 14th International Conference on Advanced Communication Technology (ICACT)*, 2012, pp. 744–749.

[2] F. Mercaldo and A. Santone, "Formal equivalence checking for mobile malware detection and family classification," *IEEE Trans. Software Eng.*, vol. 48, no. 7, pp. 2643–2657, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3067061

[3] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 11:1–11:29, 2018. [Online]. Available: https://doi.org/10.1145/3162625

[4] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Comput. Secur.*, vol. 81, pp. 123–147, 2019. [Online]. Available: https://doi.org/10.1016/j.cose.2018.11.001

[5] D. Turner, S. Entwisle, O. Friedrichs, D. Ahmad, D. Hanson, M. Fossi, S. Gordon, P. Szor, E. Chien, D. Cowings *et al.*, "Symantec internet security threat report: trends for july 2004-december 2004," *Retrieved July*, vol. 30, p. 2005, 2005.

[6] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen, "An approach for detection and family classification of malware based on behavioral analysis," in *2016 International Conference on Computing, Networking and Communications, ICNC 2016, Kauai, HI, USA, February 15-18, 2016*. IEEE Computer Society, 2016, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ICCNC.2016.7440587

[7] J. Zhu, J. Jang-Jaccard, A. Singh, I. Welch, H. AI-Sahaf, and S. Camtepe, "A few-shot meta-learning based siamese neural network using entropy features for ransomware classification," vol. 117, p. 102691, zhu-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740482200089X

[8] Y.-H. Chen, J.-L. Chen, and R.-F. Deng, "Similarity-based malware classification using graph neural networks," vol. 12, no. 21, p. 10837, chen-2. [Online]. Available: https://www.mdpi.com/2076-3417/12/21/10837

[9] K. Huang, Y. Ye, and Q. Jiang, "Ismcs: An intelligent instruction sequence based malware categorization system," in *2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*, 2009, pp. 509–512.

[10] Y. H. Park, D. S. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching," in *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop, CSIIRW 2010, Oak Ridge, TN, USA, April 21-23, 2010*, F. T. Sheldon, S. J. Prowell, R. K. Abercrombie, and A. W. Krings, Eds. ACM, 2010, p. 45. [Online]. Available: https://doi.org/10.1145/1852666.1852716

[11] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 2013, pp. 3422–3426. [Online]. Available: https://doi.org/10.1109/ICASSP.2013.6638293

[12] Y. Ye, T. Li, Y. Chen, and Q. Jiang, "Automatic malware categorization using cluster ensemble," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, Eds. ACM, 2010, pp. 95–104. [Online]. Available: https://doi.org/10.1145/1835804.1835820

[13] C. Molloy, J. Banks, H. H. Steven Ding, P. Charland, A. Walenstein, and L. Li, "Adversarial variational modality reconstruction and regularization for zero-day malware variants similarity detection," in *2022 IEEE International Conference on Data Mining (ICDM)*, pp. 1131–1136, molloy-1.

[14] S.-C. Hsiao, D.-Y. Kao, Z.-Y. Liu, and R. Tso, "Malware image classification using one-shot learning with siamese networks," vol. 159, pp. 1863–1871, hsiao-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919315595

[15] M. Conti, S. Khandhar, and P. Vinod, "A few-shot malware classification approach for unknown family recognition using malware feature visualization," vol. 122, p. 102887, conti-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822002814

[16] J. D. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Investig.*, vol. 3, no. Supplement, pp. 91–97, 2006. [Online]. Available: https://doi.org/10.1016/j.diin.2006.06.015

[17] E. G. Lazo. (2021) Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion techniques. https://www.microsoft.com/en-us/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques/.

[18] M. Botacin, V. H. G. Moia, F. Ceschin, M. A. A. Henriques, and A. Grégio, "Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios," *Digit. Investig.*, vol. 38, p. 301220, 2021. [Online]. Available: https://doi.org/10.1016/j.fsidi.2021.301220

[19] C. Molloy, S. H. H. Ding, B. C. M. Fung, and P. Charland, "H4rm0ny: A competitive zero-sum two-player markov game for multi-agent learning on evasive malware generation and detection," in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pp. 22–29, molloy-3.

[20] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *2019 IEEE Security and Privacy Workshops, SP Workshops 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 8–14. [Online]. Available: https://doi.org/10.1109/SPW.2019.00015

[21] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[22] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," vol. 48, no. 10, pp. 3771–3789, mehrotra-1.

[23] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," pp. 815–823, schroff-1. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Schroff_FaceNet_A_Unified_2015_CVPR_paper.html

[24] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole EXE," raff-1. [Online]. Available: http://arxiv.org/abs/1710.09435

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] T. Moon, "The expectation-maximization algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[27] J. M. Kleinberg, "Navigation in a small world," *Nature*, vol. 406, no. 6798, pp. 845–845, 2000.

[28] M. Boguñá, D. V. Krioukov, and K. C. Claffy, "Navigability of complex networks," *CoRR*, vol. abs/0709.0303, 2007. [Online]. Available: http://arxiv.org/abs/0709.0303

[29] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Inf. Syst.*, vol. 45, pp. 61–68, 2014. [Online]. Available: https://doi.org/10.1016/j.is.2013.10.006

[30] MalwareBazaar. (2024) MalwareBazaar. https://bazaar.abuse.ch/

[31] A. Tharwat, "Linear vs. quadratic discriminant analysis classifier: a tutorial," *Int. J. Appl. Pattern Recognit.*, vol. 3, no. 2, pp. 145–180, 2016. [Online]. Available: https://doi.org/10.1504/IJAPR.2016.079050

[32] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997. [Online]. Available: https://doi.org/10.1006/jcss.1997.1504

[33] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[34] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[35] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep convolutional malware classifiers can learn from raw executables and labels only," krcal-1.
[36] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware: Toward malware classification interpretation," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 21:1–21:29, 2021. [Online]. Available: https://doi.org/10.1145/3423096
[37] Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. C. Grundy, "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 40:1–40:38, 2021. [Online]. Available: https://doi.org/10.1145/3446905
[38] D. Zou, Y. Wu, S. Yang, A. Chauhan, W. Yang, J. Zhong, S. Dou, and H. Jin, "Intdroid: Android malware detection based on API intimacy analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 39:1–39:32, 2021. [Online]. Available: https://doi.org/10.1145/3442588
[39] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, pp. 8:1–8:28, 2020. [Online]. Available: https://doi.org/10.1145/3371924
[40] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," vol. 153, p. 102526, gibert-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303868
[41] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," koch-1.
[42] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Advances in Neural Information Processing Systems*, vol. 6. Morgan-Kaufmann, bromley-1. [Online]. Available: https://proceedings.neurips.cc/paper/1993/hash/288cc0ff022877bd3df94bc9360b9c5d-Abstract.html
[43] M. U. Demirezen, "Image based malware classification with multimodal deep learning," vol. 10, no. 2, pp. 42–59, demirezen-1. [Online]. Available: https://dergipark.org.tr/en/pub/ijiss/issue/67160/1048722
[44] F. Deldar and M. Abadi, "Deep learning for zero-day malware detection and classification: A survey," *ACM Comput. Surv.*, vol. 56, no. 2, pp. 36:1–36:37, 2024. [Online]. Available: https://doi.org/10.1145/3605775
[45] J. Gennari and D. French, "Defining malware families based on analyst insights," in *2011 IEEE International Conference on Technologies for Homeland Security (HST)*, 2011, pp. 396–401.
[46] D. Evans. (2011) The Internet of Things - How the Next Evolution of the Internet is Changing Everything. https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf
[47] J. F. Cevallos M., A. Rizzardi, S. Sicari, and A. C. Porisini, "Nero: Neural algorithmic reasoning for zero-day attack detection in the iot: A hybrid approach," *Computers & Security*, vol. 142, p. 103898, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824002001