EVOLIoT: A Self-Supervised Contrastive Learning Framework for Detecting and Characterizing Evolving IoT Malware Variants

Mirabelle Dib
mi_dib@encs.concordia.ca
Concordia University

Sadegh Torabi
storabi@gmu.edu
George Mason University

Elias Bou-Harb
elias.bouharb@utsa.edu
University of Texas at San Antonio

Nizar Bouguila
n_bouguila@encs.concordia.ca
Concordia University

Chadi Assi
assi@encs.concordia.ca
Concordia University

ABSTRACT

Recent years have witnessed the emergence of new and more sophisticated malware targeting the Internet of Things. Moreover, the public release of the source code of popular malware families such as *Mirai* has spawned diverse variants, making it harder to disambiguate their ownership, lineage, and correct label. Such a rapidly evolving landscape makes it also harder to deploy and generalize effective learning models against retired, updated, and/or new threat campaigns. In this paper, we present EVOLIoT, a novel approach aiming at combating "concept drift" and the limitations of inter-family IoT malware classification by detecting drifting IoT malware families and understanding their diverse evolutionary trajectories. We introduce a robust and effective contrastive method that learns and compares semantically meaningful representations of IoT malware binaries and codes without the need for expensive target labels. We find that the evolution of IoT binaries can be used as an augmentation strategy to learn effective representations to contrast (dis)similar variant pairs. We discuss the impact and findings of our analysis and present several evaluation studies to highlight the tangled relationships of IoT malware, as well as the efficiency of our contrastively learned feature vectors in preserving semantics and reducing out-of-vocabulary size in cross-architecture IoT malware binaries.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

IoT malware classification, concept drift, contrastive learning

ACM Reference Format:
Mirabelle Dib, Sadegh Torabi, Elias Bou-Harb, Nizar Bouguila, and Chadi Assi. 2022. EVOLIoT: A Self-Supervised Contrastive Learning Framework for Detecting and Characterizing Evolving IoT Malware Variants. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3517393

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3517393

1 INTRODUCTION

The recent growth in the number of IoT devices has motivated the rise of IoT-tailored malware that enable cyber-attacks as part of coordinated and monetized large-scale botnets [66]. The public release of the source code of some of the main IoT malware families such as the *Mirai* [2], has forever shaped the IoT threat landscape. Specifically, the effectiveness of *Mirai*, and the ability to add new exploits to the codebase has paved the way for more advanced and sophisticated *Mirai*-like malware variants such as *Hajime* [20] and *Satori* [34], to name a few. The increasing number of detected IoT malware and the threat associated with the IoT-driven cyberattacks has pushed the security community to focus their effort on deploying specialized honeypots [24, 43, 53] to detect IoT malware/botnet and investigate their inner operations [2, 11, 66]. This is essential for developing rigorous mitigation approaches against the spread of IoT malware strains by building effective learning-based malware detectors and classifiers [16, 51, 60].

Despite such effort, the complex relationship and similarities in terms of code reuse among malware variants bring several challenges for malware analysis including labeling, provenance, triage, lineage analysis, as well as family and authorship attribution. In fact, it is unclear what makes each group of malware distinct, what links together popular families, or how the same malware family evolves over time. This is also reflected by the labels assigned by anti-virus (AV) vendors, which are often inconsistent and coarse-grained, and often unable to capture the code reuse between IoT malware and their evolutionary characteristics. More importantly, the rapid evolution of the IoT threat landscape along with the characteristics of newly detected IoT malware variants in terms of their massive code reuse and underlying relationship [11, 12], is most likely to cause the performance of classifiers to degrade with time, as old malware campaigns are typically retired/updated while new ones are developed [23]. This change in the data distribution of a machine learning model is called concept drift [23], which makes it challenging to generalize existing learning models that were trained with older data to new, previously-unseen samples.

In order to build sustainable models for IoT malware classification, it is important to identify when the model shows signs of aging, by which it fails to effectively recognize new variants and adapt to potential changes in the data, especially when accounting for in-class evolution of IoT malware families. Thus, to build an effective and robust classifier, we must aim at detecting drifting IoT variants within the same malware family (i.e., in-class evolution) and consequently, interpret the meaning behind the drift to identify

which mutations distinguish one variant from another. Note that previous research relied on the prediction decisions of a learning model as a by-product of the classification process [16, 30, 58]. However, it is likely that a drifting sample that does not belong to any class will be assigned with high confidence to the wrong class (i.e., closed-world assumption), which has been previously validated.

To mitigate this confidence bias, calibrated probability predictors (e.g., Venn-Abers Predictors [14]) as well as statistical nonconformity measures [37] have been proposed. Although useful, these approaches cannot draw concrete conclusion on drifting/evolving samples and lose their effectiveness on high dimensional data. A more recent work by Yang et al. [72], used an auto-encoder coupled with a contrastive loss to compress the data and learn an effective distance between samples of different classes. While their resulting distance function can efficiently detect and rank drifting samples from distinct classes, their approach is not tailored to in-class drifting samples, which is more relevant in the context of IoT. Alternatively, a plethora of works studied the evolution of malware binaries by computing the similarity between functions extracted from the decompiled binary code (e.g., instructions), basic blocks, control flow graphs (CFGs) [17, 19, 32, 70, 73], and execution traces [4, 35]. While they provide invaluable insights, none of them is applied on Linux-based IoT malware. Cozzi et al. [12] took this opportunity to identify code similarities between IoT malware families using function-level binary diffing using off-the-shelf tools that are not tailored for IoT. Additionally, their approach required a substantial amount of manual adjustments, which hampers its scalability and feasibility for real-time threat detection and analysis.

In this work, we aim at filling this gap by detecting evolving IoT variants and understanding their evolutionary trajectories, in a systematic and scalable way tailored to the peculiarities of IoT malware. We propose EVOLIoT, a self-supervised contrastive learning approach based on pre-trained language models such as BERT [15], which effectively learns and compares semantically meaningful representations of binary code, without the need for expensive target labels. This presents an immense advantage to the problem of scarcity of labeled data (e.g., emerging IoT malware) in security applications. In fact, the proposed contrastive objective views the evolution of IoT malware binaries as a natural language augmentation strategy, which maximizes the mutual information between malware sequences and their conserved malicious function. As such, EVOLIoT identifies evolved samples by constructing a "positive" pair through feeding the same sample to the encoder twice to get two embeddings that only differ in hidden dropout masks found in Transformers [25]. As such, the model learns to predict positive pairs among other embeddings (i.e., negative pairs). In other words, EVOLIoT learns to encourage "disagreement" across evolutionary views by contrasting the rest of the embeddings, and thus, learn to discriminate between samples coming from the same class.

To this end, we make the following main contributions:

- We leverage the power of contrastive learning to address concept drift and the limitations of inter-family IoT malware classification due to the evolution of IoT malware. We present EVOLIoT, a robust and effective contrastive method that learns and compares semantically meaningful representations of IoT malware binaries without the need for expensive target labels.
- We are among the first to address the limitations of inter-family classification and attribution of IoT malware binaries using contrastive learning. We propose to view the evolution of IoT malware binaries as a desirable choice of augmentation to construct "views" for contrastive learning in a security application, from both a theoretical and technical point of view. We illustrate that our contrastive learning objective, which is based on evolutionary augmentation, directly encourages representational invariance to shared features across positive views while at the same time, encouraging disagreement across views by dealing with same-class augmentations as negative to each other.
- Motivated by the number of IoT samples, their diverse target architectures and their general lack of obfuscation, we adopt a cross-architecture code-based analysis that can capture a binary's malicious intent and evolutionary essence. Our framework leverages the cutting-edge BERT architecture [15] to deeply infer the underlying code semantics regardless of the binary's instruction set architecture (ISA). We also consider a well-balanced instruction normalization strategy that conserves as much contextual information as possible for cross-architecture syntactic variations while maintaining efficient computation.
- We evaluate our approach using a large corpus of IoT malware binaries that were detected over a course of 3 years. We leverage an interpretable strings-based analysis to detect and validate more than 50 variants belonging to the top 3 IoT malware families: *Mirai*, *Gafgyt*, *Tsunami*. Our analysis highlights the constant evolution of variations among each family from different perspectives such as the added target exploits and 0-day vulnerabilities, TOR-enabled botnet communication, and botnet behaviors (e.g., detection evasion), to name a few.
- We extensively evaluate our proposed method on three applications, by leveraging EVOLIoT as a semantic search engine for finding semantically similar variant queries, and testing the effect of our balanced normalization process and our cross-architecture embeddings in reducing out-of-vocabulary instructions and preserving semantics, respectively.
- We make our ground truth dataset with identified fine-grained labels, as well as the full list of identified evolutionary trajectories (e.g., exploits, variants) available[1] to researchers to support future work.

The rest of the paper is organized as follows. We provide background information and discuss the problem scope in Section 2. Detailed information about the proposed approach is presented in Section 3. We discuss our main findings, evaluation results, and limitations in Section 4. We present an overview of related work in Section 5 before concluding the work in Sections 6.

2 BACKGROUND AND PROBLEM SCOPE

In what follows, we elaborate on the problem of concept drift in the context of security applications (e.g., malware evolution) and highlight the power of attentive and self-supervised methods as effective solutions.

[1] https://github.com/IoTMalw/EVOLIoT

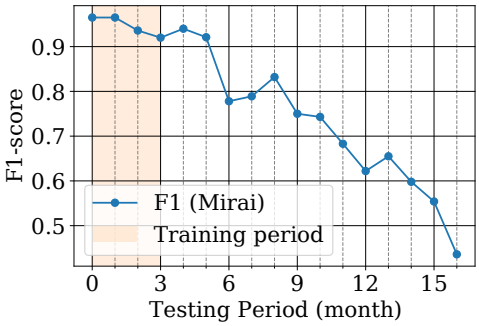


Figure 1: The impact of drifting samples on the classification accuracy for samples of the *Mirai* family (trained/tested MLP classifier with a 3 months sliding window).

2.1 Concept Drift (In-Class Evolution)

Concept drift has been used to describe the problem of the changing relation between the input data and the target variable over time [23]. In cybersecurity, these changes apply to the malware development and data generation process where attackers are constantly modifying their attack vectors, trying to bypass defenders' solutions [7]. Moreover, the evolution of malware is another problem related to this challenge, where the process of defining and improving variants results in new types of attacks. Concept drift in a multi-class classification setting usually involves drifting samples from previously unseen families (new class), or drifting samples from an existing class but with changing behavioral patterns (in-class evolution).

In this paper, we focus on the problem of *in-class evolution*, which is understudied in literature and more relevant in the context of IoT malware. Specifically, we examined the in-class evolution of samples from the *Mirai* family over one year by training an MLP classifier on samples detected in the first 3 months while testing with previously-unseen samples detected in the following 3 months intervals, respectively. As illustrated in Figure 1, we capture the impact of the drifting samples through examining the degrading accuracy of the MLP classifier over time (detailed in Appendix A). In fact, previously proposed ML-based solutions for detecting concept drift [23] are not necessarily suitable for security applications as they mainly rely on the collection of new sets of well-prepared and labelled data to statistically assess model behaviors [5, 14, 18]. However, in the context of cybersecurity, new attacks/data are usually unknown thus, it is almost impractical to assume that the incoming data is sufficiently labeled for (re)training classification models. Moreover, data labelling is usually time-consuming, expensive, and requires expertise. Instead, we focus on a more practical scenario that leverages contrastive learning, as explained in the following sub-section.

Contrastive Learning (CL). Contrastive learning is a type of Self-supervised Learning (SSL), which allows the model to learn sentence-level semantics by comparison. In general, SSL has emerged as a powerful method for learning effective representations without the need for expensive target labels [15]. This presents an immense advantage to the problem of scarcity of labeled, clean data (e.g., malware) in cyber security applications. Moreover, it helps a model gain more generalization ability by learning from large amounts of unlabeled data, in contrast to a supervised model which learns only from what is available in the training data.

To this end, we rely on contrastive learning, which works by pulling semantically close neighbors together and pushing apart non-neighbors. Recent progress on self-supervised contrastive methods has proven its effectiveness in learning good data representations for instance discrimination and semantic similarity tasks in various domains such as computer vision [8, 29], audio processing [52], and computational biology [42, 71]. Hence, we explore the idea of contrastive learning to learn semantically-aware binary code representations of IoT malware variants to detect their mutations and evolution (Section 3). Moreover, to effectively learn sentence embeddings from unlabeled data, we incorporate pre-trained Attention and Transformer language models such as BERT [15], as described in the following sub-section.

Attentive Transformer Language Model. BERT's model architecture is a multi-layer bi-directional Transformer encoder based on the original implementation described in [65], which provides rich vector representations of a natural language by capturing the contextual meanings of words and sentences using a multi-head self attention mechanism [3]. It consists of two main processes: (a) a *pre-training* phase, based on masked language model (MLM) and next sentence prediction (NSP) strategies to build a generic model that considers context and orders of words and sentences in a large data corpus; (b) a *fine-tuning* process that applies the pre-trained model to a specific downstream task. Sentence-BERT (SBERT) [56], proposed as a modification of the pre-trained BERT network, uses siamese and triplet network structures to derive semantically meaningful sentence embeddings that can be efficiently compared using cosine-similarity. In this work, we demonstrate that a contrastive objective, coupled with pre-trained language models such as BERT [15], can be extremely effective in learning sentence embeddings from unlabeled data.

2.2 Problem Scope and Insights

The widespread use of packing and obfuscation made static analysis generally inadequate in the malware domain [50]. However, to our advantage, the current number of samples and the general lack of sophisticated obfuscation in IoT binaries enable code-based analysis [11, 12]. Conveniently, modelling the binary code of an executable can provide a precise reflection of its malicious intent and evolutionary essence, especially in terms of code reuse and added functionalities. Hence, identifying evolving malware strains can be equivalent to a binary similarity comparison problem, where two samples have syntactically or semantically (dis)similar feature vectors. Specifically, the equivalent semantics of a binary code can be defined as a sequence of instructions that carries out an identical task from a logical function in the original source. Yet, the realm of IoT presents a few challenges:

(1) Multiple efforts are in place to ensure that IoT malware samples can run on devices with diverse hardware architectures and system configurations. Meaning, two binaries compiled from the same source code for different architectures (e.g., ARM, MIPS) can present syntactically different instructions.
(2) In contrast to previous work [19, 21, 32, 44, 70], the code equivalence problem cannot be resolved at the *function*-level due to

the tangled relationships of similarities and code reuse between IoT malware binaries. In fact, "stolen" code components from a function or a set of functions can be inserted into other code, that is, borrowed code is not necessarily a function.

(3) Identifying the start of functions within binary code is a common problem in the context of static malware analysis. In fact, the performance of well-known analysis tools such as IDA Pro has been shown to deteriorate significantly (drops to 60%) when identifying the start of functions in stripped binaries [13], whereas it works consistently for correctly identifying instructions and basic blocks.

In fact, the key to identifying differing IoT malware variants is to explain their evolutionary trajectories regardless of their target instruction set architectures (ISAs). Therefore, considering the abovementioned challenges, determining the similarity between two malware variants requires a precise and efficient cross-architecture embedding model, which can capture the semantics and dependencies of instructions.

While previous solutions failed to procure such results, we proceed by regarding instructions as words and basic blocks as sentences. Particularly, we present a new method to train such sentence embeddings without relying on training data, and by leveraging Sentence-Transformer and contrastive learning. The idea is to consider the evolution of IoT binaries as an augmentation strategy, and hence encode the same instance twice to form a *positive* pair. The distance between these two embeddings will be minimized, while the distance to other embeddings of the other sentences in the same batch will be maximized (i.e., they serve as *negative* pairs). An overview of our proposed approach is detailed in Section 3.

3 APPROACH

In this work, we propose a new approach to address the problems of concept drift and intra-family IoT malware classification by leveraging contrastive learning. We have two main objectives: (i) Detecting in-class evolving/drifting IoT malware binaries, and (ii) interpreting the meaning behind the drift. To achieve our objectives, we follow a multi-stage methodology, as illustrated in Figure 2. In particular, we detect drifting IoT binaries by extracting a feature modality from the malware binaries (e.g., assembly code), normalizing all instructions, learning a vector to represent the semantic meaning of the assembly code of the executable binary, and then deriving an effective distance function to measure the dissimilarity of samples. Second, the goal of interpreting the drift is to identify the causes of the drift (e.g., added functionalities, behavior changes, etc.) and link the detection decision to semantically meaningful features.

In what follows, we provide further details about each stage of the proposed methodology (Figure 2).

3.1 Feature Extraction & Pre-Processing

We start our analysis by extracting and normalizing important features (e.g., assembly instructions) that carry enough information to effectively differentiate a drifting sample from another.

Formally, the assembly code of an executable is a set of basic blocks each containing a sequence of assembly instructions, denoted by $I_f : (i_1, i_2, i_3, ..., i_m)$, where m is the number of instructions in the function. These sequences of machine instructions are analogous to a natural language, which implies the possibility of utilizing effective techniques in an NLP domain such as BERT for a binary task. In fact, the authors of InnerEye [73] apply the idea of Neural Machine Translation (NMT) to a binary function similarity comparison task by regarding instructions as words and basic blocks as sentences.

We start by extracting a set of essential artifacts ❶ (e.g., feature modalities) from our malware samples (Table 1), using commercially popular static binary analysis tools. We leverage IDA Pro [31] to extract basic blocks of assembly instructions. Following that, we perform *Instruction Normalization* to map instructions to various tokens, which are leveraged by most deep learning methods to generate input sequences for their training phase. Hence, we propose a well-balanced instruction normalization ❷ that is neither too generic nor too specific, to avoid an *out-of-vocabulary* (OOV) problem, while capturing code semantics with tokens that hold rich information. In fact, various approaches [17, 19, 44, 73] have adopted mechanical conversions where the most common one is replacing every immediate operand with a single notation such as `immval`, without a thorough consideration of their contextual meanings. Such a coarse-grained normalization renders every call instruction identical, hence loses a considerable amount of contextual information. However, retaining immediate values [44] can raise an OOV problem due to a massive number of unseen instructions (tokens).

Conveniently, to maintain contextual and semantic information, good word embeddings must rely on an individualized normalization strategy where binary code semantics are expressed as precisely as possible while using a reasonable number of tokens. For instance, it is important to differentiate between immediate values because an immediate can represent either a call invocation, a memory reference to jump to, a string reference, a statically-allocated variable, etc. Therefore, erasing such differences makes an embedding rarely distinguishable from another.

To this end, combining successful normalization strategies from the literature, we make sure to: (i) differentiate between a jump and a call destination, a string value, or a memory reference, (ii) consider different sizes in a 32-bit register [19], (iii) keep stack pointers or base pointers intact [44] while preserving pointer expressions to maintain memory access information. Finally, Opcodes are not normalized and are retained as they are.

3.2 Instruction Embedding Model

To this end, we leverage a pre-trained language model, namely BERT [15], to encode the normalized assembly instructions ❸ and then fine-tune all the parameters using the contrastive learning objective (§3.3). Note that BERT takes as input a token "sequence" which can either be single sentence or a pair of sentences packed together. We consider a sentence to be a block of assembly instructions. Such input representation allows BERT to handle a variety of downstream tasks and thereby later serves our contrastive objective (§3.3). We leverage pre-trained checkpoints of BERT with a number of settings/considerations, as described in Appendix B.

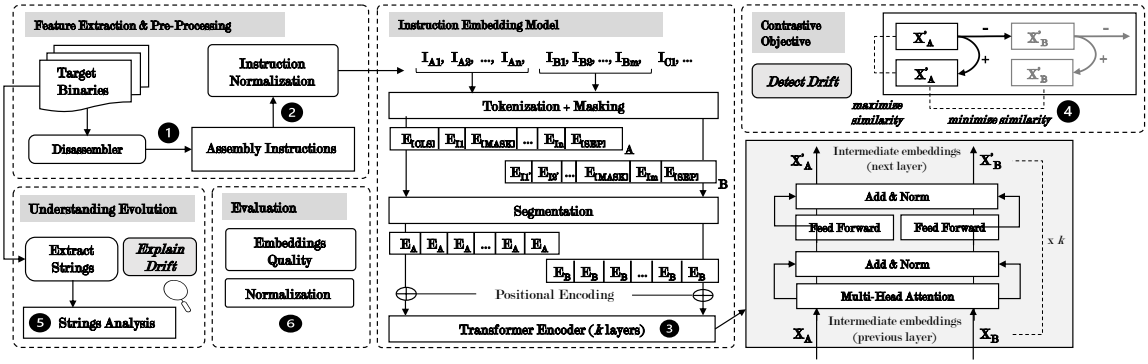


Figure 2: An overview of the proposed EVOLIoT framework/approach and its various stages.

3.3 Contrastive Objective

In this section, we present the contrastive embedding framework ❹ behind EVOLIoT, which can produce superior "sentence" embeddings (i.e., assembly instructions feature vectors) from incoming unlabeled malware data, and learn how to compare them to identify differing variants within the same family. The idea of contrastive learning is to learn a good representation of unlabeled data by distinguishing similar samples from the others. It assumes a set $D = \{x_k^+\}$ including paired examples x_i and x_j which are semantically related, also referred to as *positive* pairs. Then, a neural network base encoder $f(.)$ extracts representation vectors from the data examples, denoted as $z_i = f(x_i)$ and $z_j = f(x_j)$. As described in §3.2, we adopt the pre-trained language model BERT as our encoder. Moreover, the contrastive objective is to identify x_j in the set of negative examples $\{x_k^-\}_{k \neq i}$ for a given x_i. Let $sim(z_i, z_j) = \frac{z_i^\top z_j}{||z_i||.||z_j||}$ be the cosine similarity between two feature vectors, the contrastive loss is defined as [8]:

$$\ell_{i,j} = -\log \frac{\exp(sim(z_i, z_j) \,/\, \tau)}{\sum_{z_k \in Z^-} \exp(sim(z_i, z_k) \,/\, \tau)}, \tag{1}$$

where $\tau \in \mathbb{R}$ denotes a temperature hyper-parameter ($\tau = 0.05$) to adjust the scaling of the similarity scores, and $Z^- := \{z_k\}_{k \neq i}$. This loss, which is computed across all positive pairs (i, j) and (j, i) in a mini-batch, brings the anchor and positive samples together while driving the anchor and negative samples apart.

In this work, we consider the evolution of IoT binaries over the years as a theoretically and technically desirable augmentation strategy to construct "views" of the input. As described in Appendix C, IoT malware variants can be considered as "evolutionary augmented views" of a common ancestor x (e.g., first *Mirai* variant to appear), while $\mathcal{T}$ can denote possible evolutionary trajectories characterized by changing features and mutations (Figure 9). The key idea is that properties of the ancestral sequence will be preserved in both descendants (i.e., views). Therefore, by training a contrastive encoder to project them to nearby locations in the latent space, their proximity is ensured to correspond to similar malicious functions, even without explicit labels.

In general, contrastive learning encourages "agreement" between important features across evolutionary views. In contrast, our contrastive objective aims at identifying factors that contribute instead to the "disagreement" between evolutionary views. In particular, while the existing contrastive schemes act by pulling all augmented samples toward the original sample, we suggest to additionally push the samples with shifting transformations away from the original.

To learn a good alignment for positive pairs and identify the evolved samples is to construct two different embeddings as *"positive pairs"* by feeding the same sample twice to the encoder and getting two embeddings that only differ in hidden *dropout* masks (as detailed in Appendix C). Thus, the distance between these two embeddings will be minimized, while the other embeddings in the same batch will serve as *"negative"* examples, and the model will predict the positive one among negatives. By contrasting the rest of the embeddings, the model will learn to discriminate between samples coming from the same class, by maximizing the distance to the shifting embeddings instead of minimizing it.

3.4 Understanding Evolutionary Changes

In this section, we try to elaborate on the *interpretability* by comprehending what has caused a sample to drift from its neighbors over time. Once our contrastive module has identified groups of evolving variants, it will output a label for each sample, representing the cluster to which it belongs. Given the fact that the model's performance is very tightly coupled with the representations used (i.e., assembly instructions), and while such raw features are very informative for the learning model, there are often not very human-interpretable by themselves.

Therefore, to understand the evolutionary characteristics of evolving IoT malware, we perform a strings-based similarity analysis on the binary samples, which would allow us to attribute the variant changes to more interpretable features. As such, we utilized reverse-engineering techniques and static malware analysis to extract meaningful strings that provide clues about the suspect malware and its functionalities (e.g., attack commands, target devices, malicious payloads, C&C IP addresses, unique strings, etc.). In particular, we use regular expressions to obtain special textual indicators such as IP addresses associated with possible adversarial hosts, and distinctive keywords associated with unique variants or known malicious commands to search for target devices, exploited vulnerabilities and attack operations. An example of the extracted strings from a malware binary is presented in Listing 1 in Appendix D.

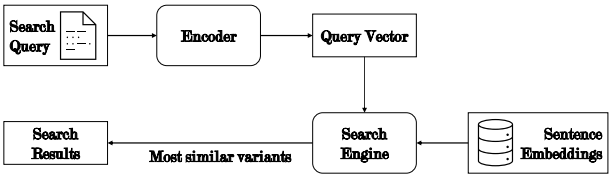


Figure 3: Overview of our semantic code search engine.

3.5 Evaluation of Instruction Embeddings

We evaluate the quality of our generated cross-architecture code embeddings in terms of their ability to preserve useful semantics information as compared to other baselines. Our qualitative analysis consists of showing that our code embedder can be efficiently used as a semantic search engine, as shown in Figure 3, for finding known variants in our unsupervised dataset with high precision, as well as learning semantic information about syntactically different instructions. In addition, we assess the effect of our well-balanced normalization process in reducing *out-of-vocabulary* instructions when presented with previously unseen and diverse instructions sets. All evaluation results are presented in Section 4.4.

4 RESULTS

In this section, we examine the impact of the rapid evolution of IoT malware on the performance of classifiers over time (§A), followed by an assessment of the impact of contrastive learning on the detection of in-class drifting IoT malware binaries.

4.1 Data Collection

In this paper, we leverage well-known online malware repositories such as VirusShare [67] and VirusTotal [68] along with a specialized IoT honeypot (IoTPOT [6]) to obtain over 90,000 IoT malware samples that were detected between 2018 and 2021. For consistency purposes, we performed pre-processing steps to filter out corrupted or non-executable files (e.g., HTML/ASCII files), ending up with 74,429 IoT malware binaries. To label these samples, we have retrieved their VirusTotal (VT) analysis reports and processed them with AVClass [59], which determines the most likely family name attributed to malware samples by applying a majority rule on reported labels from multiple anti-virus engines. Table 1 reports the top identified families, with *Mirai* and *Gafgyt* dominating the dataset. Such imbalance in the data across different families is a mere reflection of the monopoly inflicted by *Mirai* and its descendants on the IoT threat landscape. In fact, the effectiveness of the *Mirai* family motivates adversaries to reuse/recycle the Mirai source code, with most of the "new" IoT botnets to represent mere modifications of the *Mirai* code base. Moreover, the analysis of Internet-scale scanning activities generated by infected IoT devices confirms the prevalence of *Mirai*-like malware in the wild [27, 55, 62].

It is worth noting that 34% of the collected samples, AVClass [59] failed to reach a consensus for a common family name, as 2,664 of them were not associated with known IoT malware families to anti-virus engines (*Unknown*), and 24,271 were never found in VirusTotal reports (*Unseen*). This is an indication that the identified malware binaries can be either new, or have not been detected yet by anti-virus vendors. In fact, it is unrealistic to assume that security analysts are aware of all malware families deployed in the wild. Yet, it stands to confirm the effectiveness of honeypots towards a promptly collection of IoT malware samples.

Table 1: Distribution of malware by family.

Label	Count (%)
Mirai	40,974 (55.05)
Gafgyt	3,976 (5.34)
Tsunami	956 (1.28)
Dofloo	464 (0.62)
Others	122 (0.16)
Unknown	2,664 (2.23)
Unseen	24,271 (32.60)
Total	74,429 (100)

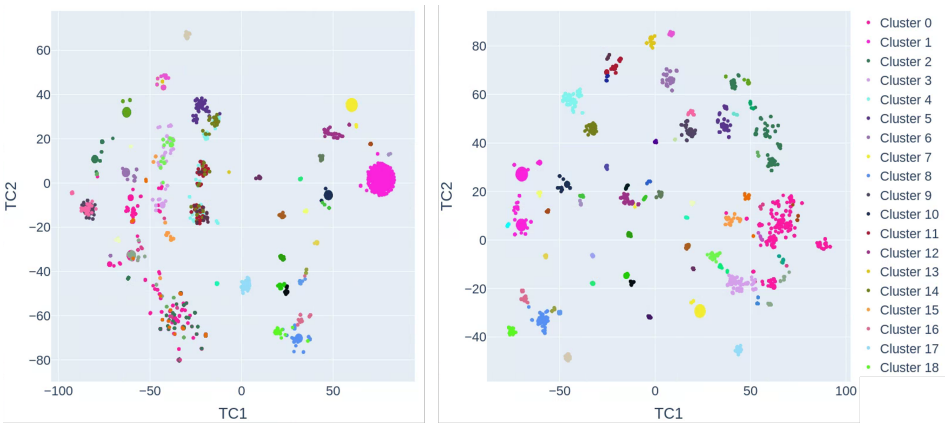
Note that all the collected IoT malware data is available for research purposes and can be directly requested from the above-named sources (e.g., IoTPOT [53]). Unfortunately, the restricted sharing policies instilled by the data providers prevents us from directly sharing the analyzed samples with the research community.

4.2 Impact of Contrastive Objective

To examine the impact of using contrastive learning to identify drifting variants within the same class, we first evaluate the quality of the obtained clusters of variants. A good clustering method will produce high quality clusters in which the intra-cluster similarity is low and the inter-cluster similarity is high. The *Silhouette index* is a measure of how similar an object is to its own cluster (*cohesion*) compared to other clusters (*separation*) [33]. A score $s \in [-1, 1]$ is calculated for each object, where '1' indicates that this is a perfectly clustered object. Values near 0 indicate overlapping clusters while negative values generally indicate that a sample has been assigned to the wrong cluster. Further details about the calculation of the silhouette score is presented in Appendix E.

To illustrate the impact of the contrastive learning objective, we compare the learned representations on 6,000 randomly selected *Mirai* samples using Standard and the proposed Contrastive Embedding. As shown in Figure 4, we empirically visualize the representations learned by our contrastive module and their distinctive separation in the latent space. We use t-SNE [64], which is a non-linear dimensionality reduction technique for visualizing data in a low 2-dimensional space. We observe that the contrastive loss leads to better variants separation, making it easier to distance samples from others. In fact, the contrastive objective leads to better data clustering than the standard embedding model, with a high average *silhouette score* (about 0.89) for the total number of identified clusters. Moreover, we observe fewer overlapping samples when comparing Figures 4.a and 4.b and a clearer distinction between them. It is interesting to see the dense pink cluster in Figure 4.a further dissected when the contrastive objective is applied in Figure 4.b. In fact, our strings-based analysis in Section 4.3 confirms the presence of two similar variants in this cluster that only differ by one additional exploit targeting GPON routers. This demonstrastes the effectiveness of EVOLIoT in distinguishing fine-grained modifications in evolved variants.

Comparison with State-Of-Art (CADE [72]). We conduct an experiment to evaluate the performance of the open-sourced CADE



(a) Standard (b) Contrastive

Figure 4: t-SNE visualization of learned representations on 6,000 randomly selected *Mirai* samples with (a) Standard embedding, and (b) Contrastive Embedding.

Table 2: Drifting detection results on the Drebin and IoT malware datasets when comparing CADE with a baseline vanilla autoencoder (AE).

Method	Drebin (Avg)				IoT (Avg)				
	Precision	Recall	F1	Norm. Effort	Precision	Recall	F1	Norm. Effort	Insp. Count
Vanilla AE	0.63	0.88	0.72	1.48	0.53	0.99	0.69	2.54	22050
CADE	0.96	0.96	0.96	1.00	0.84	0.80	0.82	1.35	9941

[72] when applied to our IoT malware samples. The authors' objective is to detect and interpret drifting samples from previously unseen malware families by similarly leveraging the power of contrastive learning. By design, CADE uses an auto-encoder augmented with contrastive loss to learn compressed representation of the training data. The first term of their contrastive loss minimizes the reconstruction loss of the auto-encoder while the second term minimizes the Euclidean distance between two samples in the latent space, if they are from the same class. To evaluate the drifting sample detection module, the authors pick one of the families as the *previously unseen family*, while the other families are split into training and testing sets. In this respect, we select three IoT malware families (*Mirai*, *Gafgyt*, *Tsunami*) to form a balanced dataset of their assembly code artifacts, and pick one of the 3 families as the unseen. As such, the unseen family is not available during training and the goal is to correctly identify samples belonging to the hidden family as drifting samples in the testing time. Given a ranked list of detected samples, CADE calculates three evaluation metrics: precision, recall and F_1 score. In addition to these metrics, CADE ranks drifting samples based on their distance to the nearest centroids to focus on those that are furthest away.

As shown in Table 2, CADE is compared with a baseline vanilla autoencoder (AE) without contrastive loss. Additionally, we evaluate the performance of CADE when applied to IoT malware samples. For each experiment (choice of previously unseen family), we report the highest F_1 score for each model. The "*inspecting effort*" is a metric that refers to the total number of inspected samples to reach the reported F_1 score, normalized by the number of true drifting samples in the testing set. A higher inspecting effort means that more analysis is required to manually verify if a sample truly belongs to the unseen family in the testing set.

For each evaluation metric, we report the mean value across all settings, as well as the normalized inspecting effort. As summarized in Table 2, for the IoT malware dataset, the number of samples ("inspection count") that need to be validated by security analysts as truly belonging to the previously unseen family is very high (compared to 600 inspected on Drebin by the authors), yet still lower with CADE, which confirms the importance of contrastive learning. Moreover, the obtained results strongly suggest that CADE performs well in most settings (i.e., using the Drebin dataset), but not as well when applied to the IoT malware dataset. This is a limitation of CADE, especially when testing their technique on malware mutations/variants within the same family. As such, CADE is primarily focused on *Type A* concept drift (i.e., introduction of a new class in a multi-class setting), while we address its limitations in identifying drifting samples that are from existing classes (i.e., *Type B*: in-class evolution).

Further Comparison. Several invaluable works have been proposed for clustering IoT malware families using static and dynamic features [63], as well as for dissecting and in-depth studying a singular family such as *Mirai* [2, 66]. Yet, to the best of our knowledge, we are among the first to detect in-class drifting IoT malware binaries and study their mutations over time. In fact, Cozzi et al.[12] proposed a code-level clustering and function similarity solution to draw the lineage of IoT malware families. By design, the authors leveraged a popular off-the-shelf binary diffing tool to perform a detailed code similarity analysis on 93k samples that have appeared between 2015 and 2018, and discovered shared components across different families. While a direct comparison between our works

is not feasible, it is clear that their approach is sophisticated and limited by the stripped nature of IoT malware binaries, and by the scalability and direct application of binary diffing tools to IoT binaries.

Moreover, multiple approaches on binary code similarity have been proposed to study malware lineage inference [12, 32, 35, 41, 48], however they are only applied in the context in which they were developed and not on Linux-based IoT malware. If they do, they only support the MIPS and ARM architectures. In addition, most proposed solutions compute similarity between binaries from their execution traces [4, 35], which are too-coarse grained for variant identification, or at the function or CFG levels [12, 17, 32, 44, 70], which depend on the ability of finding the start of functions. This is inherently difficult in the context of IoT due to the stripped nature of binaries. Cozzi et al. [12] try to circumvent this by propagating known symbols in unstripped binaries to stripped binaries.

4.3 Characterizing Variant Changes

Using our contrastive objective, we identified 44 variants of *Mirai* and 11 variants of *Gafgyt*, as highlighted in Figure 5. However, it is still unclear how the same family evolved over time, what makes these variants different, and whether or not samples from different families are connected. To answer these questions, we take a step further by extracting and investigating their *strings*, which will shed more light on the common/differing characteristics of the identified variants.

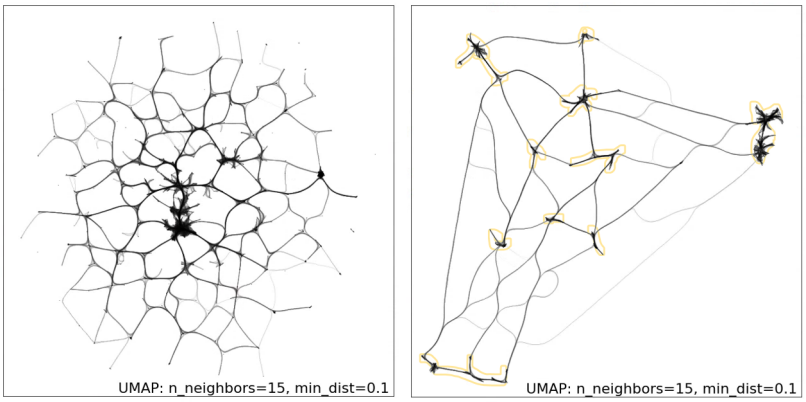
Given the extracted strings, we perform a string-based similarity analysis on the identified malware samples. As illustrated in figure 5, we draw the connectivity plot for 10,000 *Mirai* and 3,000 *Gafgyt* samples. The darker edges are indicators of a high similarity between the samples. By looking at the detection time of the samples, we noticed that the older *Mirai* variants are likely to share more resemblance and appear at dense and more central areas, whereas newer *Mirai* variants are growing apart and appear farther at the edges. In addition, it is clearly observed that the *Mirai* variants are more closely connected, forming more visibly connected regions. On the other hand, *Gafgyt* samples seem to be less connected (lower similarities) and thus, placed further apart. We proceed by understanding these differences and the threats associated with each identified cluster of variants.

Among the three most populated and closely related *Mirai* clusters (#1, #2 and #3), we found 7,124 samples, that were first scanned in 2018, exploiting two vulnerabilities affecting GPON routers. When these two vulnerabilities are used in conjunction, they enable the execution of commands sent by an authorized remote attacker to a vulnerable device. What makes these kindred clusters separate, is their expansion to target a vulnerability in Huawei HG532 routers (#2) and Netgear routers (#3). As such, among samples in Cluster #9, we found 97 attributed to the *Apep botnet*, which aside from dictionary attacks via telnet, also spread through infected Huawei routers, which explains why they are closely-connected to Cluster #2. By investigating further, we also uncovered samples belonging to the *Xjno* variant in the same cluster, using the same command & control URL as the *Apep* variant. This stands to confirm that EVOLIoT identifies fine-grained characteristics shared among different samples.

In addition, we found 1,917 samples spread across Clusters #5 and #11, taking advantage of a ThinkPhP vulnerability which allows them to breach web servers using the PHP frameworks via dictionary attacks on default credentials to gain remote access to them. Among those in Cluster #11, we detected a more recently scanned variant, using a new exploit to infect Huawei devices with malware named after the recent Covid pandemic. Moreover, we found 7 samples belonging to the *Miori* variant in Cluster #8 spreading through a remote code execution in ThinkPHP. They start by contacting other IP addresses using Telnet, while also listening on port 42352 for commands from their C2 servers. Next, they verify whether a targeted device was successfully infected by sending the command "/bin/busybox MIORI". Interestingly, we found one evolved *Miori* sample in Cluster #11 also downloading a new malicious payload named "corona3.sh" and killing a list of competing botnets to ensure persistence. As such, by grouping together covid-related samples in Cluster #11, EVOLIoT sheds light on the rapid evolution of IoT malware towards exploiting global events for malware propagation and distribution. Additionally, Cluster #22 contains complementary samples to Cluster #5, belonging to the *Yowai* variant and spreading using the same ThinkPHP exploit, however, we found references to commands for killing competing botnets that might have infected the targeted device. Among their kill list are the names of 58 variants and the majority are unknown to us. This proves the existence of a multitude of (unknown) IoT variants that attackers are familiar with, which gives them a bigger advantage over the control of the next wave of cyber attacks.

Furthermore, while 70 samples belonging to the *Omni* variant were found in Cluster #1 targeting GPON routers, samples belonging to the same variant were found in a further away cluster, #27. This is because they are leveraging a total of 11 vulnerabilities associated with multiple target devices. While the identified vulnerabilities are publicly known, this is the first variant using all 11 exploits in conjunction. This evolved campaign of *Omni* variants is preventing further infection of infected devices by dropping packets received on 15 different ports using the *iptables* command. Interestingly enough, this variant shares the same IP address for downloading payloads and reporting to the C&C server with 194 *Gafgyt* samples, which explains why they are nearly clustered. This is either a case of mislabeling or an indication that the same bot master is controlling two independent IoT campaigns concurrently. By looking closely at them, we found that they share the same exploits as the *Mirai Omni* variant except for an OS command injection in the UPnP SOAP interface which renders 5 types of D-Link routers vulnerable. However, newer samples of this *Gafgyt* variant detected in 2019 have incorporated a command injection exploit targeting D-Link DSL-2750B routers. As such, while *Mirai* and *Gafgyt* might not share the same codebase, they still exploit and target common devices, using similar attack methods and reporting to the same C2.

Moreover, we performed the same analysis on some *Mirai* samples that have been clustered on the edges, further than the rest, and found that they have been first scanned by VirusTotal between 2020 and 2021. We found 38 samples in Cluster #19 that seem to be spreading by hijacking a vulnerability in digital video recorders (DVR) provided by KGUARD, as well as connecting to their C2 via the Tor-Proxy protocol using 7 different ports. We found embedded



(a) Mirai **(b) Gafgyt**

Figure 5: A weighted graph constructed using UMAP [45] representing the connectivity between the strings embeddings of (a) 10,000 *Mirai* and (b) 3,000 *Gafgyt* samples with highlighted nodes that represent 11 variants identified by EVOLIoT (§4.3).

URLs with the ".onion" extension. Out of curiosity, we looked at Cluster #29 represented by the singly connected node in Figure 5.a, and found indications of the string *"aurora"*. EVOLIoT has pushed it further than other *Mirai* variants because it is spreading using a 0day vulnerability in the Ruijie (NBR700) routers. Interestingly, that same vulnerability is being exploited by an older *Mirai* variant which has appeared two months earlier, which explains the single connection edge. However, the vulnerability exploit payload in the *aurora* variant uses many empty variables with confusing names to distract security analysts. It additionally has a mechanism to check whether it is running in a sandbox environment, by verifying the path and filename where the sample is located. This is an indication that *Mirai* variants are growing to be more resilient. In addition, while the sample is not packed, a lot of sensitive information are hidden and seem to be encrypted using a different algorithm than *Mirai*'s simple XOR encryption.

4.4 Evaluation

To evaluate the quality of our instructions embeddings, we first assess the effectiveness of our model in learning code semantics by finding semantically similar samples in our dataset, compared to other approaches. Then, we evaluate the quality of our cross-architecture instructions in preserving semantics versus syntactics. Finally, we assess the power of normalization in reducing the instructions vocabulary size while capturing code semantics with tokens that hold rich information.

Semantic-Search Engine. To evaluate the quality of our obtained embeddings, our first task aims at finding the most semantically related code from a collection of candidate codes. This would allow us to verify the effectiveness of our model in learning code semantics information. As such, we relied on our strings-based analysis §4.3 to build our own ground truth data for evaluation. This dataset consists of a 587 samples of *Mirai* belonging to 6 different variants: *Omni* (70), *Apep* (97), *Yowai* (37), *Satori* (72), *Dark* (91), *Josho* (220). To identify samples that belong to the same variant class, we randomly pick a sample per variant class to encode it using our embedding model and retrieve all other semantically similar code embeddings in our dataset. To do so, we leverage FAISS [36] with Approximate Nearest Neighbor. FAISS uses principal component analysis to reduce the number of dimensions in the vectors, to reduce the computation when comparing a query vector against already embedded vectors. Next, it partitions the data into similar clusters to compare the query vector against these partition/cluster centroids. Once the nearest centroid is found, only full vectors within that centroid are compared to the query, and all others are ignored. Hence, the complexity of the required search area is significantly reduced.

We evaluate the performance of our search engine with varying retrieval thresholds to inspect whether true positives are ranked at the top. We sort the returned results and evaluate each of them in sequence. We collect recall and precision at top-k position ($k = 10$). Recall is the fraction of the documents that are relevant to the query that are successfully retrieved, while the precision is the fraction of the documents retrieved that are relevant to the user's information need. It is trivial to achieve recall of 100% by returning all documents in response to any query. Therefore, recall alone is not enough but one needs to measure the number of non-relevant documents also, for example by computing the precision. In fact, since each query can have multiple relevant results, we use Mean Average Precision (MAP) as the metric to evaluate the code search on our embeddings dataset. The mean average precision for a set of queries is the mean of the average precision scores for each query, which can be calculated as $MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$, where Q is the number of queries in the set and $AveP(q)$ is the average precision for a given query q. We use 4 baselines for comparison of the evolution results (Word2vec [47], Doc2vec [39], Code2vec [1], and Bi-LSTM [61]). Please refer to Appendix F for further details about the used baselines.

It is important to note that several other solutions [17, 32, 44] have been proposed to analyze binary code similarity and semantic code retrieval, however, we cannot directly compare them to our approach since they mainly rely on the availability of the source code along with binary functions and/or extracted control flow graphs, which are not considered in our approach.

Besides the above-mentioned baselines that rely on code embeddings, we also manually select features to compare with basic-block feature-based machine learning classifiers such as Gemini [70] and Genius [22]. We compare our embedding model to the SVM classifier using 16 manually selected features (as detailed in Table 4 in

Table 3: Results of the semantic search retrieval using different baselines as code embedders.

Model	Performance (MAP)
Word2vec	0.36
Doc2vec	0.39
Code2vec	0.57
Bi-LSTM	0.74
EVOLIoT	0.89

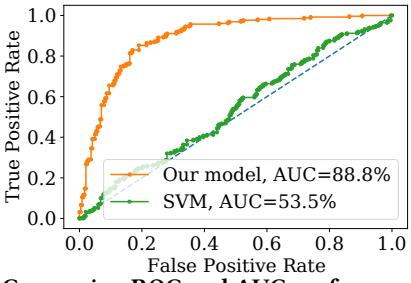


Figure 6: Comparing ROC and AUC performance results.

Appendix F) from binary disassembly (e.g., number of instructions, average basic blocks, etc.). We leverage an ELF binary analysis service developed by [11] which evaluates ELF binaries in a multi-architecture sandboxing environment using static and dynamic malware analysis techniques to extract these features. We refer the reader to our public link for a complete list of extracted features.

As shown in Table 3, we were able to efficiently retrieve, validate, and label using our semantic search engine, previously unknown variants in our dataset with a mean average precision of 89%. This confirms that our embedding model generate semantically similar/relevant code embeddings, and outperforms different baselines. Moreover, our model outperforms the SVM classifier trained on the manually selected features, and achieves much higher AUC values, as illustrated in Figure 6. This is because manually selected features cause significant information loss in terms of the contained instructions and the dependencies between these instructions, while our model precisely encodes and preserves the block semantics across different variants.

Cross-Architecture Instruction Embeddings. In this section, we present our results from qualitatively analyzing the instruction embeddings for different architectures. Zuo et al. [73] have shown that instructions compiled for the same architecture cluster together while those compiled from a different architecture cluster far from each other. This is due to the syntactic variations that an architecture introduces, which creates further challenges in the context of IoT due to the diverse nature of cross-platform devices.

Our objective is to evaluate the quality of our cross-architecture instruction embedding model, by picking variants that have been clustered together for their semantic relationship, and analyzing their target architectures. For our embedding model to be effective, two semantically similar yet syntactically different binaries should still be clustered closely in the space. As such, we first use t-SNE [65] to plot the instruction embeddings in a two-dimensional plane. As shown in Figure 7, our analysis of clusters #1, #3, and #14 show that semantically related samples are clustered together even if they have different target instruction architectures. In fact, cluster #1 contains semantically similar samples targeting a variety of CPU

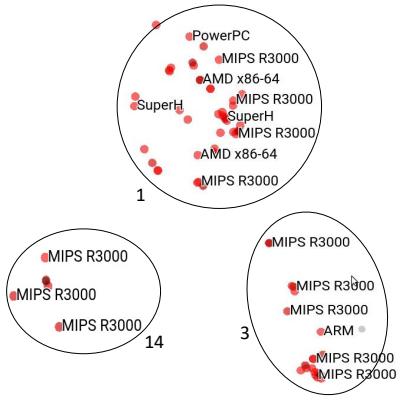


Figure 7: Visualization of syntactically different yet semantically similar embeddings belonging to clusters 1, 3, and 14.

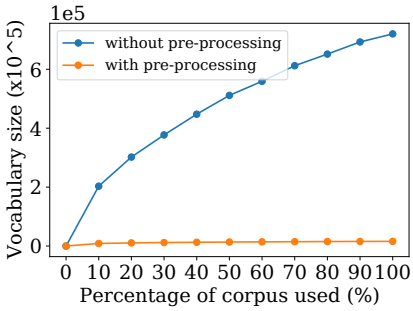


Figure 8: Visualization of the growth of the vocabulary size when the corpus size increases.

architectures, while clusters #3 and #14 are only reserved to the top two most popular architectures among IoT devices. Such variability is explained by evolving samples that are expanding the pool of potential devices which can be compromised.

Effectiveness of Well-balanced Normalization. A preprocessing normalization step is applied to avoid an out-of-vocabulary problem, while preserving code semantics instructions with tokens that hold rich information. To evaluate the impact of instructions normalization, we seek to understand whether the instructions vocabulary size is affected with or without pre-processing. As such, we disassembled 4,385 *Mirai* variants that have been scanned in 2019 and counted the total number of assembly instructions: 80,299,331. Then, we divided the corpus in 10 equal sized parts to see how the vocabulary size grows in terms of the percentage of analyzed corpus. As clearly observed in Figure 8, the vocabulary size grows significantly and relatively fast without pre-processing, while remains relatively small when pre-processing is applied. As shown by the analysis results, EVOLIoT can be leveraged as a semantic search engine for finding semantically similar variants while outperforming previous approaches, in addition to preserving semantics in cross-architecture embeddings and reducing out-of-vocabulary instructions.

4.5 Limitations and Future Work

This work has a number of limitations, which may hamper the generalizability and the validity of the reported findings. For instance, we rely on the fact that IoT malware samples are still largely

unobfuscated as compared to generic malware. However, we were unable to extract useful strings from around 23% of the analyzed samples. Despite that, our analysis showed that the majority of these samples did not employ sophisticated obfuscation, thus, can be de-obfuscated using off-the-shelf tools (e.g., UPX). Furthermore, due to the lack of fine-grained IoT malware labels and lack of well-designed ground truth datasets for evaluation, we built our own ground truth to evaluate the proposed semantic search engine (§4.4). As such, while the dataset might be relatively small and not representative, it represents a reliable dataset since the samples were carefully examined and labelled manually through our strings-based analysis.

For future work, we intend to address the above mentioned limitations by investigating different techniques for malware de-obfuscation and unpacking to account for a more representative sample of IoT malware while extending the work to non-IoT malware space, which observes more obfuscation. Further, we will combine our manual analysis with information obtained from online threat repositories to obtain a more representative ground truth that can improve our evaluation outcomes. Finally, the unveiling of diverse IoT malware variants inspires us to study in future work the competition and coordination among IoT botnets in the wild, which is still underexplored in literature.

5 RELATED WORK

IoT Threat Landscape The public release of the source code of the *Mirai* botnet [57] has prompted new actors to easily bootstrap their own botnet and compete over the control of vulnerable IoT devices. Antonakakis et al. [2] have presented the first comprehensive study of the *Mirai* botnet and described the effect of the shared source code on the release of new specialized variants. Several other works have focused on the customizations of *Mirai* (e.g., Hajime, BrickerBot) to study the change in their infection behavior [38, 66] as well as to reveal shared password combinations used during brute forcing [10]. While these works focus on one botnet, Griffioen et al. [27] have focused on multiple *Mirai*-like variants competing for the same IoT devices to identify the differences in success between botnets. They exploit 7,500 IoT honeypots and a flaw in the design of *Mirai*'s random number generator to conclude that IoT botnets are not self-sustaining. Alternatively, Torabi et al. [62] leveraged passive network measurements collected from the darknet along with IoT device information to infer compromised IoT devices in the wild. Additionally, several studies [53, 66] have deployed IoT-tailored honeypots to capture a wide range of emerging IoT threats and their characteristics. While previous works hold valuable insights on the threat landscape of IoT malware, further research is needed to study the dynamics behind the emergence of new intra-family strains and track the evolution of existing ones.

Malware Evolution and Lineage Inference. Malware is constantly evolving to adapt to survival needs, bug fixes, and feature additions. Lineage studies are most useful when applied to malware as version information is usually not available [28, 35]. Inspired by the evolution of species and molecules, Goldberg et al. [26] and iLINE[35] produced malware phylogeny trees using a directed acyclic graph (DAG). Lindorfer et al. [41] investigated the malware evolution process by mapping API calls to disassembled code in order to identify mutations in the malware family. Calleja et al. [6] identified code reuse between benign software and different Windows malware families observed over a period of 40+ years. Their observations ranged from common utility functions to anti-detection routines to credentials for brute-forcing attacks.

Moreover, several techniques for binary similarity gained momentum as they can be applied for malware lineage inference. In a recent survey, Haq et al. [28] highlighted the strengths and weaknesses of 61 approaches on binary code similarity, including those used for malware evolution [32, 35, 41, 48]. BEAGLE was proposed by Lindorfer et al. [41] to study malware evolution by comparing binary code in terms of API calls extracted using behavioral analysis. Huang et al. [32] identified code reuse in two Windows malware families by computing the similarity between functions extracted from binaries at the instruction, basic block and CFG levels, while Jang et al. [35] have combined low-level binary features, code-level basic blocks and binary execution traces for lineage construction. Existing solutions have been designed around computing CFGs and matching procedures that cannot be adapted to compute a constant size signature of a binary on which a similarity measure can be applied. They also cannot be immediately extended to cross-platform similarity and therefore cannot be applied on Linux-based IoT malware. Cozzi et al. [12] took this opportunity to identify code similarities between IoT malware families using function-level binary diffing. However, they resorted to popular off-the-shelf binary diffing tools not tailored for IoT, which required a substantial amount of manual adjustments and validation.

Recent advancements in machine learning techniques have seen effective applications in binary code similarity. Ding et al. [17] have recently proposed an assembly clone search approach named *Asm2Vec*, which learns a vector representation of the sequence of instructions executed on a certain path of the CFG. While it outperformed several state of-the-art solutions in the field of binary similarity, Asm2Vec is not directly applicable for semantic clones across architecture as it only generates single-platform embeddings, and it has the performance overhead of performing random walks on CFGs. In another work, Xu et al. [70] proposed *Gemini*, a neural network-based approach to compute binary function embeddings based on an *annotated CFG*, a graph containing manually selected features. However, their annotation approach based on manual feature selection can introduce human bias, by preferring, for instance, arithmetic instructions over others. In *InnerEye* [73], Zuo et al. apply the idea of Neural Machine Translation (NMT) to find similar CFG blocks. However, it is not clear how such embeddings are a representation of entire functions. Secondly, the used LSTM architecture is burdened by long term dependencies/memorization, hence does not cope well with long sequences of instructions [40]. *DeepBinDiff* [19] has been proposed to find differences between two binaries by leveraging deep neural networks and greedy graph matching. Yet, DeepBinDiff requires call symbols and strings which would be stripped away or obfuscated in statically linked malware, hence it is a single architecture solution.

6 CONCLUSION

In this paper, we present a robust, accurate, and semantic-aware assembly instructions representation generator, EVOLIoT, which

leverages the evolution of IoT binaries as effective augmentation strategy for contrastive learning. Our approach achieves both efficiency and accuracy for cross-architecture assembly instructions search without relying on any expensive (e.g., CFG) or manually selected features. Additionally, we addressed the problem of in-class concept drift by detecting evolving IoT malware variants and interpreting the reason behind their drift. Further, we comprehensively evaluate the effectiveness and robustness of our proposed approach with a large corpus of IoT malware data. Our findings shed light on the evolving IoT threat landscape characterized by the ever-lasting *Mirai* variant, which is spreading by using new undisclosed vulnerabilities, persisting by killing other bots and looking out for sandbox environments, encrypting its communication using Tor proxies, and incorporating encryption algorithms.

REFERENCES

[1] Uri Alon, Meital Zilberstein, et al. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Manos Antonakakis, Tim April, et al. 2017. Understanding the Mirai botnet. In *26th USENIX Security Symposium*. USENIX Association, Vancouver, BC, 1093–1110.

[3] Dzmitry Bahdanau, Kyunghyun Cho, et al. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations*. ICLR, Banff, Canada, 1–15.

[4] Ulrich Bayer, Paolo Milani Comparetti, et al. 2009. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium*, Vol. 9. NDSS, San Diego, CA, 8–11.

[5] Albert Bifet and Ricard Gavalda. 2007. Learning from Time-Changing Data with Adaptive Windowing. In *Proceedings of the 2007 SIAM international conference on data mining*. SIAM, Minneapolis, Minnesota, 443–448.

[6] Alejandro Calleja, Juan Tapiador, et al. 2018. The Malsource Dataset: Quantifying Complexity and Code Reuse in Malware Development. *IEEE Transactions on Information Forensics and Security* 14, 12 (2018), 3175–3190.

[7] Fabrício Ceschin, Marcus Botacin, et al. 2019. Shallow Security: On the Creation of Adversarial Variants to Evade Machine Learning-based Malware Detectors. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*. ACM, Vienna, Austria, 1–9.

[8] Ting Chen, Simon Kornblith, et al. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119. PMLR, Virtual, 1597–1607.

[9] Joana Costa, Catarina Silva, et al. 2014. Concept Drift Awareness in Twitter Streams. In *2014 13th International Conference on Machine Learning and Applications*. IEEE, Detroit, MI, 294–299.

[10] Andrei Costin and Jonas Zaddach. 2018. IoT Malware: Comprehensive Survey, Analysis Framework and Case Studies. *BlackHat USA* 1, 1 (2018), 1–9.

[11] Emanuele Cozzi, Mariano Graziano, et al. 2018. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, 161–175.

[12] Emanuele Cozzi, Pierre-Antoine Vervier, et al. 2020. The Tangled Genealogy of IoT Malware. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, Austin, USA, 1–16.

[13] Ahmad Darki, Michalis Faloutsos, et al. 2019. IDAPro for IoT Malware analysis?. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET19)*. USENIX Association, Santa Clara, CA, 1–15.

[14] Amit Deo, Santanu Kumar Dash, et al. 2016. Prescience: Probabilistic Guidance On the Retraining Conundrum for Malware Detection. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*. ACM, Vienna, Austria, 71–82.

[15] Jacob Devlin, Ming-Wei Chang, et al. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. ACL, Minneapolis, MN, USA, 4171–4186.

[16] Mirabelle Dib, Sadegh Torabi, et al. 2021. A Multi-Dimensional Deep Learning Framework for IoT Malware Classification and Family Attribution. *IEEE Transactions on Network and Service Management* 18, 2 (2021), 1165–1177. https://doi.org/10.1109/TNSM.2021.3075315

[17] Steven H. H. Ding, Benjamin C. M. Fung, et al. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, 472–489. https://doi.org/10.1109/SP.2019.00003

[18] Denis Moreira dos Reis, Peter Flach, et al. 2016. Fast Unsupervised Online Drift Detection Using Incremental Kolmogorov-Smirnov Test. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, San Francisco, California, 1545–1554.

[19] Yue Duan, Xuezixiang Li, et al. 2020. Deepbindiff: Learning Program-Wide Code Representations for Binary Diffing. In *Network and Distributed System Security Symposium*. NDSS, San Diego, California, 1–16.

[20] Sam Edwards and Ioannis Profetis. 2016. Hajime: Analysis of A Decentralized Internet Worm for IoT Devices. *Rapidity Networks* 16 (2016), 1–18.

[21] Sebastian Eschweiler, Khaled Yakdan, et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Network and Distributed System Security*, Vol. 52. NDSS, San Diego, California, 58–79.

[22] Qian Feng, Rundong Zhou, et al. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna, Austria, 480–491.

[23] João Gama, Indrė Žliobaitė, et al. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4 (2014), 1–37.

[24] Usha Devi Gandhi, Priyan Malarvizhi Kumar, et al. 2018. HIoTPOT: Surveillance on IoT Devices Against Recent Threats. *Wireless personal communications* 103, 2 (2018), 1179–1194.

[25] Tianyu Gao, Xingcheng Yao, et al. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Empirical Methods in Natural Language Processing*. EMNLP, Punta Cana, Dominican Republic, 1–17.

[26] Leslie Ann Goldberg, Paul W Goldberg, et al. 1998. Constructing Computer Virus Phylogenies. *Journal of Algorithms* 26, 1 (1998), 188–208.

[27] Harm Griffioen and Christian Doerr. 2020. Examining Mirai's Battle over the Internet of Things. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual, 743–756.

[28] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (2021), 38 pages.

[29] Kaiming He, Haoqi Fan, et al. 2020. Momentum Contrast for Unsupervised Visual Representation Learning. In *2020 Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Seattle, WA, 9729–9738.

[30] Dan Hendrycks and Kevin Gimpel. 2017. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. In *5th International Conference on Learning Representations*. ICLR, Toulon, France, 1–12.

[31] Hex-Rays. 2005. IDA Pro Disassembler. https://www.hexrays.com/products/ida/.

[32] He Huang, Amr M. Youssef, et al. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, Abu Dhabi, United Arab Emirates, 155–166.

[33] Félix Iglesias, Tanja Zseby, et al. 2019. Absolute Cluster Validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 9 (2019), 2096–2112.

[34] Vijayan J. 2018. Satori Botnet Malware Now Can Infect Even More IoT Devices.

[35] Jiyong Jang, Maverick Woo, et al. 2013. Towards Automatic Software Lineage Inference. In *22nd USENIX Security Symposium*. USENIX Association, Washington, DC, 81–96.

[36] Jeff Johnson, Matthijs Douze, et al. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547.

[37] Roberto Jordaney, Kumar Sharad, et al. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *26th USENIX Security Symposium*. USENIX Association, Vancouver, BC, Canada, 625–642.

[38] Constantinos Kolias, Georgios Kambourakis, et al. 2017. DDoS in the IoT: Mirai and Other Botnets. *Computer* 50, 7 (2017), 80–84.

[39] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *International Conference on Machine Learning (ICML'14)*. PMLR, Beijing, China, II–1188–II–1196.

[40] Zhouhan Lin, Minwei Feng, et al. 2017. A Structured Self-Attentive Sentence Embedding. In *5th International Conference on Learning Representations*. ICLR, Toulon, France, 1–15.

[41] Martina Lindorfer, Alessandro Di Federico, et al. 2012. Lines of Malicious Code: Insights into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, Orlando, Florida, 349–358.

[42] Amy X Lu, Haoran Zhang, et al. 2020. Self-Supervised Contrastive Learning of Protein Representations by Mutual Information Maximization. *BioRxiv* 1, 1 (2020), 1–17.

[43] Tongbo Luo, Zhaoyan Xu, et al. 2017. Iotcandyjar: Towards an Intelligent-Interaction Honeypot for IoT Devices. *Black Hat* 1 (2017), 1–11.

[44] Luca Massarelli, Giuseppe Antonio Di Luna, et al. 2019. Safe: Self-Attentive Function Embeddings for Binary Similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Gothenburg, Sweden, 309–329.

[45] L. McInnes, J. Healy, et al. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *ArXiv e-prints* 1, 1 (2018), 1–63. arXiv:1802.03426

[46] Yu Meng, Chenyan Xiong, et al. 2021. Coco-lm: Correcting and Contrasting Text Sequences for Language Model Pretraining. *NeurIPS* abs/2102.08473 (2021), 1–16.

[47] Tomas Mikolov, Ilya Sutskever, et al. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 3111–3119.
[48] Jiang Ming, Dongpeng Xu, et al. 2015. Memoized Semantics-based Binary Diffing with Application to Malware Lineage Inference. In *IFIP International Information Security and Privacy Conference*, Vol. AICT-455. Springer, Hamburg, Germany, 416–430.
[49] Aziz Mohaisen and Omar Alrawi. 2014. Av-meter: An Evaluation of Antivirus Scans and Labels. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Egham, UK, 112–131.
[50] Andreas Moser, Christopher Kruegel, et al. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, Miami Beach, FL, USA, 421–430.
[51] Quoc-Dung Ngo, Huy-Trung Nguyen, et al. 2020. A Survey of IoT Malware and Detection Methods Based on Static Features. *ICT Express* 6, 4 (2020), 280–286.
[52] Aaron van den Oord, Yazhe Li, et al. 2018. Representation Learning with Contrastive Predictive Coding. *CoRR* abs/1807.03748 (2018), arXiv:1807.03748.
[53] Yin Minn Pa Pa, Shogo Suzuki, et al. 2016. IoTPOT: A Novel Honeypot for Revealing Current IoT Threats. *Journal of Information Processing* 24, 3 (2016), 522–533.
[54] Feargus Pendlebury, Fabio Pierazzi, et al. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification Across Space and Time. In *28th USENIX Security Symposium*. USENIX Association, Baltimore, MD, 729–746.
[55] Morteza Safaei Pour, Antonio Mangino, et al. 2020. On Data-Driven Curation, Learning, and Analysis for Inferring Evolving Internet-of-Things (IoT) Botnets in the Wild. *Computers & Security* 91 (2020), 101707.
[56] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. ACL, Hong Kong, China, 3982–3992.
[57] Mirai Source Code Release. 2016. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/.
[58] Konrad Rieck, Thorsten Holz, et al. 2008. Learning and Classification of Malware Behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Berlin, Heidelberg, 108–125.
[59] Marcos Sebastián, Richard Rivera, et al. 2016. Avclass: A Tool for Massive Malware Labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, Cham, 230–253.
[60] Jiawei Su, Danilo Vargas Vasconcellos, et al. 2018. Lightweight Classification of IoT Malware Based on Image Recognition. In *2018 IEEE 42Nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, Tokyo, Japan, 664–669.
[61] Kai Sheng Tai, Richard Socher, et al. 2015. Improved Semantic Representations from Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*. ACL, Beijing, China, 1556–1566.
[62] Sadegh Torabi, Elias Bou-Harb, et al. 2022. Inferring and Investigating IoT-Generated Scanning Campaigns Targeting a Large Network Telescope. *IEEE Transactions on Dependable and Secure Computing* 19, 1 (2022), 402–418. https://doi.org/10.1109/TDSC.2020.2979183
[63] Sadegh Torabi, Mirabelle Dib, et al. 2021. A Strings-Based Similarity Analysis Approach for Characterizing IoT Malware and Inferring Their Underlying Relationships. *IEEE Networking Letters* 3, 3 (2021), 161–165. https://doi.org/10.1109/LNET.2021.3076600
[64] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605.
[65] Ashish Vaswani, Noam Shazeer, et al. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., Long Beach, USA, 5998–6008.
[66] Pierre-Antoine Vervier and Yun Shen. 2018. Before Toasters Rise Up: A view Into the Emerging IoT Threat Landscape. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, Heraklion, Crete, Greece, 556–576.
[67] VirusShare. 2012. VirusShare. https://virusshare.com/
[68] VirusTotal. 2004. VirusTotal. https://www.virustotal.com/
[69] Zhuofeng Wu, Sinong Wang, et al. 2020. Clear: Contrastive Learning for Sentence Representation. *CoRR* abs/2012.15466 (2020), 1–10.
[70] Xiaojun Xu, Chang Liu, et al. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Dallas, Texas, 363–376.
[71] Kevin K Yang, Zachary Wu, et al. 2018. Learned Protein Embeddings for Machine Learning. *Bioinformatics* 34, 15 (2018), 2642–2648.
[72] Limin Yang, Wenbo Guo, et al. 2021. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In *30th USENIX Security Symposium*. USENIX Association, Virtual, 2327–2344.
[73] Fei Zuo, Xiaopeng Li, et al. 2018. Neural Machine Translation Inspired Binary Code Similarity Comparison Beyond Function P[airs. In *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS, San Diego, California, 1–15.

A OBSERVING CONCEPT DRIFT

In a real-life setting, it is unwise to assume that the data is independent and identically distributed (i.i.d.) [23], therefore, it is most likely that the model will become obsolete when the distribution of incoming data at test-time is different from that of training data. In order to confirm the presence of evolving samples in our dataset, we observe whether such samples impact the accuracy of classifiers. Particularly, we analyse the performance change (i.e., decrease) of a model over time when predicting newly collected data.

To do this, we adopt a sliding time window approach [9] where we begin by splitting the dataset into a training subset T_r with a time window size W_r, and a testing subset T_s with a time window size W_s. For a realistic setting, we enforce the following constraints: **(1)** $W_s > W_r$ to evaluate the long-term performance and robustness to decay of the classifier, **(2)** every window size is split into equal time slots of size z, to allow for a considerable and equal number of samples in each test window $[t_i, t_i + z]$, **(3)** all the samples in T_r must be precedent to the ones in T_s; violating this constraint will bias the evaluation by including future knowledge in the classifier, and **(4)** all the evaluations must assume that labels y_i of the samples $s_i \in T_s$ are unknown, even though we have their labels.

Figure 1 captures the performance of a classifier C trained on T_r and tested for each time frame $[t_i, t_i + z]$ of the testing set T_s. We chose a multilayer perceptron (MLP), which is a class of feed-forward artificial neural network (ANN), as our classifier C. Our T_r consists of a random set of samples that have been first scanned by AV engines during the first three months of the year 2018. We first test our classifier on a smaller set of samples which the classifier has not been trained on. Then we test it on samples which have been scanned during subsequent intervals of three months until 2019. As observed in Figure 1, the classifier performs well when tested with data that appeared during the same time interval (training period), with an average accuracy above 96%. Subsequently, when we start adding previously-unseen testing samples that have appeared in later months, the overall accuracy significantly drops to below 50%. This is indicative of the changing nature of variants within the same class, and as such, it is necessary to detect such drifting samples.

It is important to note that several mitigation approaches [37, 54] have been proposed to reduce the performance decay of classifiers when tested on previously-unseen or drifting data, such as retraining on different timestamps, or quarantining drifting samples and retraining the classifier solely on them. Moreover, as our objective is not to test the robustness of classifiers over time nor propose mitigation approaches, we will leave further evaluation of various classifiers and different datasets for future work.

B INSTRUCTION EMBEDDING USING BERT

We start from pre-trained checkpoints of BERT:

- We include a special token at the beginning of every sequence (`[CLS]`) to indicate the start of a sequence. In addition, we differentiate sentence pairs packed together into a single sequence by separating them with a special token (`[SEP]`). For instance,

([CLS] $W_1 W_2$ [MASK] ... W_n [SEP] $W_1 W_2$ ... [MASK] W_m), where W is a word in a sentence.

- We adopt BERT's original masked language model (MLM), which masks a percentage (e.g., 15 %) of the input tokens at random, and then predicts those masked tokens during pre-training. An advantage of masking is forcing the transformer to remember the context representation for every input token while hiding the words that it will be asked to predict at the final layer.
- BERT includes a next-sentence prediction (NSP) task which allows it to learn relationships between sentences by predicting if the next sentence in a pair is the true next or not. However, the semantics of a function should be considered *location-independent* from that of its adjacent functions. As such, attackers can copy-/borrow a certain function from another source code and add it anywhere in their code. Therefore, we do not use the NSP task of BERT.

C EVOLUTION AS CHOICE OF "VIEWS"

The critical question that follows is how to select "views" of the input, i.e., how to construct (x_i, x_j) positive pairs. In visual representations, Chen et al. [8] constructed such positive pairs $x_i = T_1(x)$ and $x_j = T_2(x)$ by taking two independent augmentations or "views" of a query image x from a pre-defined family of transformations T, where $T_1, T_2 \sim T$. Some frequently used image transformations are rotation, cropping or flipping. Recently, language and sentence representations have adopted augmentation techniques such as word deletion, substitution, and reordering [46, 69]. Moreover, in most cases, (x_i, x_j) are collected from supervised datasets. However, when it comes to malware binaries, data augmentation is inherently difficult because of their distinct nature, and incoming malware data is mostly unlabeled.

In this work, we argue that the evolution of IoT binaries over the years serves as a theoretically and technically desirable augmentation strategy to construct views. As outlined in Figure 9, in conceptualizing evolution as an augmentation plan, IoT malware variants can be considered as "evolutionary augmented views" of a common ancestor x (e.g., first *Mirai* variant to appear), while $\mathcal{T}$ can denote possible evolutionary trajectories characterized by changing features and mutations. Much as x_i and x_j can be seen as two malware variants sampled from the same family at different times. For instance, x_i can be a variant of the *Mirai* family that have appeared in 2018, while x_j can be a variant of the same family that have appeared in 2020.

The key idea is that properties of the ancestral sequence will be preserved in both descendants (i.e., views). Therefore, by training a contrastive encoder to project them to nearby locations in the latent space, their proximity is ensured to correspond to similar malicious functions, even without explicit labels. Therefore, contrastive learning directly encourages representational invariance to shared features across variants. As in it encourages "agreement" between important features across evolutionary views. Our contrastive objective is to identify what contributes instead to the "disagreement" between evolutionary views.

Hence, the next important question is how to adapt the contrastive learning objective to identify differing IoT variants within

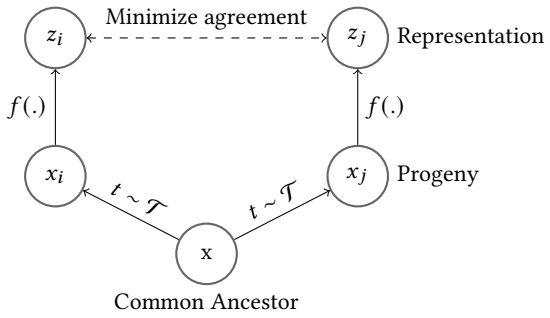


Figure 9: Re-casting SimCLR [8] as a phylogenetic tree where augmentations are the evolved malware variants.

the same family, i.e., how to encourage "disagreement" across evolutionary views. In particular, while the existing contrastive schemes act by pulling all augmented samples toward the original sample, we suggest to additionally push the samples with shifting transformations away from the original. Namely, instead of considering evolved variants as positive to each other, we attempt to consider them as negative if they belong to the same malware family. The aim is to learn robust semantic representations that can capture small input variations between variants belonging to the same class.

Learning a good alignment for positive pairs. Considering the unsupervised nature of our dataset, an effective solution is to take a collection of input samples $\{x_i\}_{i=1}^N$ and use $x_i^+ = x_i$. The key idea is that the evolutionary trajectories or mutations within the same malware family are unknown, specially since ELF binaries do not contain a timestamp of when they were compiled as opposed to generic malware. Therefore, one might rely on public forum discussions or on the VirusTotal first submission time as ground truth. However, online discussions are time-consuming and difficult to track and might not contain accurate or reliable information. On the other hand, anti-virus engines' scan time of the binaries might not coincide with the time they actually appeared in the wild, as malware can go a long time before being detected.

Hence, one way to identify the evolved samples is to construct two different embeddings as *"positive pairs"* by feeding the same sample twice to the encoder and getting two embeddings that only differ in hidden *dropout* masks, found in Transformers [25]. We denote $z_i^h = f_\theta(x_i, h)$, where h is random mask for dropout. Thus, the distance between these two embeddings will be minimized, while the other embeddings in the same batch will serve as *"negative"* examples, and the model will predict the positive one among negatives. By contrasting the rest of the embeddings, the model will learn to discriminate between samples coming from the same class, by maximizing the distance to the shifting embeddings instead of minimizing it.

Predicting the input sentence itself with only *dropout* used as noise has been shown to greatly outperform training objectives such as predicting next sentences, discrete data augmentation (e.g., word deletion and replacement) and even matches supervised training objectives [25]. We verify the effectiveness of our contrastive objective for the detection of in-class evolution in our experimental results in Section 4.2, by observing how samples that are closer to each other, form tighter groups in the latent space, making it

```
rm -rf %s; pkill -9 %s; killall -9 %s;
cd /tmp || cd /var/run || cd /dev/shm || cd /mnt;
rm -f *; /bin/busybox wget http://AnonIP/bins.sh;
chmod 777 bins.sh; sh bins.sh;
/bin/busybox tftp -r tftp.sh -g AnonIP;
```

Listing 1: Example of readable strings extracted from a malware sample with anonymized IP addresses (AnonIP).

easier to separate/distance them from other variants, as well as by comparing with previous works.

D EXTRACTED STRINGS FROM MALWARE BINARIES

Listing 1 represents an example of the extracted strings from a malware sample, which is designed to download and execute a malicious file (*bins.sh*) from a possible adversarial C&C server (*http://AnonIP/*). In addition, we notice that the malware is trying to eliminate all running processes (e.g., *rm, pkill, killall*) in order to fortify itself, and is trying different commands to download malicious payloads in case one of them fails, as seen in this consequent instruction using the TFTP protocol (e.g., *tftp -r tftp.sh -g AnonIP*). We also focus on studying the relationships and cross-family agreement between samples, which might be a reflection of reused coding practices among IoT malware families, or a case of mislabeling by anti-virus engines which is quite probable [49, 59].

E SILHOUETTE SCORE

Given an object x_i of a cluster C, the silhouette score is calculated using the following equation:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{max\{a(x_i), b(x_i)\}}, \tag{2}$$

where $a(x_i)$ is the average distance or dissimilarity of an object x_i to all other objects in the same cluster, and $b(x_i)$ is the minimum average dissimilarity of x_i to all other clusters that are not its cluster. The final index is obtained by averaging the scores for all objects in the dataset. Figure 10 is a diagrammatic representation of the above-mentioned silhouette coefficient formula.

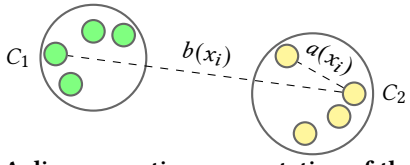


Figure 10: A diagrammatic representation of the silhouette coefficient formula $s(x_i)$. C_1 and C_2 are clusters.

F BASELINE FOR COMPARISON

We use the following baselines for comparison:

- Word2vec [47] is a popular technique to learn word embeddings using shallow neural networks. The continuous bag-of-words model (CBOW) is a method of word2vec that uses words around a target word as context. In our case, we compare by considering each token (opcode or operand) as word and instructions around each token as its context. DeepBinDiff [19] and Asm2vec [17] both leveraged a variation of word2vec based on CBOW and Paragraph Vector Distributed Memory (PV-DM) respectively, to learn embeddings of assembly functions represented as a control flow graphs (CFGs).
- Doc2vec [39] creates a numeric representation of a document of words, regardless of its length. Distributed Bag of Words (DBOW) is doc2vec algorithm where the paragraph vectors are obtained by training a neural network on the task of predicting a probability distribution of words in a paragraph given a randomly-sampled word from the paragraph. Here, a paragraph represents a code snippet or a basic block of assembly instructions.
- Code2vec [1] is a model that learns distributed representations of code called code embeddings, to evaluate its performance against the task of semantically searching code snippets. It decomposes code fragments to a collection of paths using Abstract Syntax Trees (ASTs) and learns the atomic representation of aggregated paths.
- Bi-LSTM [61] is a bidirectional sequence processing model that consists of two LSTMs: one taking the input in a forward direction, and the other in a backwards direction. Bi-LSTMs effectively increase the amount of information available and hence improve the context of a word. We treat code simply as sequences of tokens and use the neural machine translation (NMT) baseline (i.e. a 2-layer Bi-LSTM) proposed in [73] for a cross-architecture basic-block comparison.

Table 4: List of manually extracted features from binaries disassembly using Padawan ELF tool [11].

Feature	Description
average_bytes_function	Average size in bytes of a function
average_basic_blocks	Average number of basic blocks with respect to functions
average_cyclomatic_cl	Average cyclomatic complexity with respect to functions
average_location	Average lines of code with respect to functions
branch_instruction	Number of branch instructions
bytes_function	Total size in bytes of the functions
call_instructions	Number of call instructions
function_location	Percentage of instructions belonging to functions
indirect_branch_instr	Number of indirect branch instructions
max_basic_blocks	Max basic blocks
max_cyclomatic_cl	Max cyclomatic complexity
num_funcs	Number of functions detected
percent_load_covered	Percentage of covered load segment
percent_text_covered	Percentage of covered text section
syscall_instructions	Number of syscall instructions