SCRUTINIZER: Detecting Code Reuse in Malware via Decompilation and Machine Learning

Omid Mirzaei[1(✉)], Roman Vasilenko[2], Engin Kirda[1], Long Lu[1], and Amin Kharraz[3]

[1] Northeastern University, Boston, USA
{o.mirzaei,e.kirda,l.lu}@northeastern.edu
[2] VMware, Boston, USA
rvasilenko@vmware.com
[3] Florida International University, Miami, USA
ak@cs.fiu.edu

Abstract. Growing numbers of advanced malware-based attacks against governments and corporations, for political, financial and scientific gains, have taken security breaches to the next level. In response to such attacks, both academia and industry have investigated techniques to model and reconstruct these attacks and to defend against them. While such efforts have been all useful in mitigating the effects of modern attacks, automated malware code reuse inspection and campaign attribution have received less attention.

In this paper, we present an automated system, called SCRUTINIZER, to identify code reuse in malware via a novel machine learning-based encoding mechanism at the function-level. By creating a large knowledge base of previously observed and tagged malware campaigns, we can compare unknown samples against this knowledge base and determine how much overlap exists. SCRUTINIZER leverages an unsupervised learning approach to filter out irrelevant functions before code reuse detection. It provides two valuable capabilities. First, it identifies ties between an unknown sample and those malware specimens that are known to be used by a specific campaign. Second, it inspects if specific tools or functionalities are used by a campaign. Using SCRUTINIZER, we were able to identify 12 samples that were previously unknown to us and that we were able to correctly assign to well-known APT campaigns.

Keywords: Malware analysis · Code reuse detection

1 Introduction

Recent reports show that advanced malware-based breaches are increasing in volume and impact [68]. Companies are frequently becoming the targets of financially and politically-motivated attackers. For example, very recently, several biotech research firms experienced crippling targeted attacks during the COVID19 pandemic [14,54]. Despite more attention on advanced attacks recently [25,27,28,30,40,48] and their significant effects [3], there has been little investigation into how to automatically detect and identify specific advanced malware campaigns at scale when looking at large volumes of incoming malware samples.

The security community has extensively investigated how unknown attacks can be detected based on code similarity. Previous work studied binary and source code similarity testing detection [18,63,71], clone detection [4,5,35,37,42] and (fuzzy) hashing [38,41,58,67]. Very recently, the community has also begun to explore the use of modern machine learning approaches to detect more sophisticated malware attacks such as APTs [27]. While these efforts have been useful in identifying some forms of attacks, the challenge of detecting more interesting samples (e.g., APTs) in large volumes of incoming malware data is still an open challenge. Existing approaches are inadequate for characterizing malicious code reuse and campaign attribution in modern malware attacks for two primary reasons: First, relevant data about advanced (e.g., APT) samples is scarce. Hence, in real-world deployments, this limitation makes common machine learning techniques such as supervised machine learning significantly less effective in finding specific types of malware. Second, current static code similarity testing approaches are ineffective in locating previously unknown threats due to the intense use of evasive techniques by malware authors, rendering almost all of the existing static analysis approaches ineffective in practice (and hence, making dynamic analysis often necessary).

The core insight behind of our work is that almost all malware-based attacks (including advanced, targeted attacks such as APTs) follow practically proven patterns to deliver the actual malicious payload. Malware is rarely written from scratch and often depends on an existing code base or on specific, unique code bases (e.g., Mimikatz). Thus, if we accurately create a knowledge base of a large corpus of known, malicious source code snippets (some of them being known APT activity) observed in modern malware attacks, we can then compare unknown samples against this dataset and identify code similarities.

One major challenge when analyzing malware samples is the lack of source code. Hence, the malware binary needs to be analyzed and understood. While binary comparisons between malware samples are possible, in practice, even small differences between the files can lead to major reported differences even though the samples might belong to the same campaign. To be able to build an effective, accurate code similarity detection approach that works on real-world malware binaries, in the first step, we need to use dynamic analysis techniques that can execute the code and then create process snapshots of the running sample. In the second step, we need to be able to reconstruct the corresponding high-level source code from the memory dumps, and automatically locate segments of the code that are highly likely used in the malicious operation. This second step is more challenging, and has not been addressed to date.

In this work, we partnered with a well-known anti-malware company to run and extract run-time memory snapshots from 12,450 real-world malware samples.

We then built tools to analyze the process snapshots and to reconstruct the source code of the actual malicious payloads, and retrieve the corresponding function code snippets. Armed with a large set of decompiled code samples from a wide-range of malware specimens, in this paper, we propose a novel encoding mechanism to perform automated code similarity analysis. We developed a python-based framework to post-process run-time memory snapshots, and extract source code in order to be able to translate an arbitrary malware sample into a set of encoded functions that are representative of the functionality of the code. To achieve this, we take advantage of advances from the field of document classification by treating each function in the decompiled code as a sequence of words. We incorporate Siamese Neural Networks (SNNs) [9] to vectorize the functions and build the encoding model to be able to perform similarity testing. We applied our proposed similarity testing approach on 44,015 recent benign and malicious samples we received from the anti-malware company. The knowledge base we generated contains 1,734,992 clusters of extracted functions that were determined to be similar.

The automated code similarity comparison system that we built, called SCRUTINIZER, was used to analyze 3,000 random samples submitted to the anti-malware company in the summer of 2020. We could automatically identify and assign unknown samples to already known APT campaigns using function-level similarities. Evaluating advanced malware (e.g., APT) in the real-world is a great challenge because of the lack of ground truth. However, we were able to manually verify our detection results for 12 different, unknown samples by comparing our findings with those of other security vendors. Also, on average, SCRUTINIZER was able to discard on average 56% of the code of the unknown samples because it was determined to be uninteresting. Hence, SCRUTINIZER also proved to be useful during binary analysis as a filtering mechanism.

Contributions. In summary, this paper makes the following main contributions:

– We propose a machine-learning-based system, called SCRUTINIZER, to detect code reuse in malware samples. SCRUTINIZER relies on an unsupervised learning algorithm to filter out less relevant functions to malicious code samples.
– To our knowledge, we are the first to extract and use high-level source code from adversarial malware samples for performing automated malware campaign attribution. We were able to identify 12 samples that were previously unknown to us and that we were able to correctly assign to well-known APT campaigns.
– We have created a large knowledge base which contains more than 1.7M clusters of function encodings obtained from a large-scale analysis of 44K real-world advanced malware and benign samples. Both filtering and code reuse detection rely on this knowledge base to discard noisy functions and to identify code reuse.
– We release our code, our findings and other relevant information to foster the research in this area [64].

2 Approach

In this section, we describe our approach for performing automated code analysis. Building of a solution that serves this goal requires extracting the unpacked version of a given payload for decompilation, developing an efficient encoding mechanism on decompiled code for similarity testing, and constructing an unsupervised model to run code analysis on an unknown sample. The overall architecture of the proposed system for both the modeling and testing phases is shown in Fig. 1. In this section, we describe SCRUTINIZER's architecture in detail and our approach to implement each part.

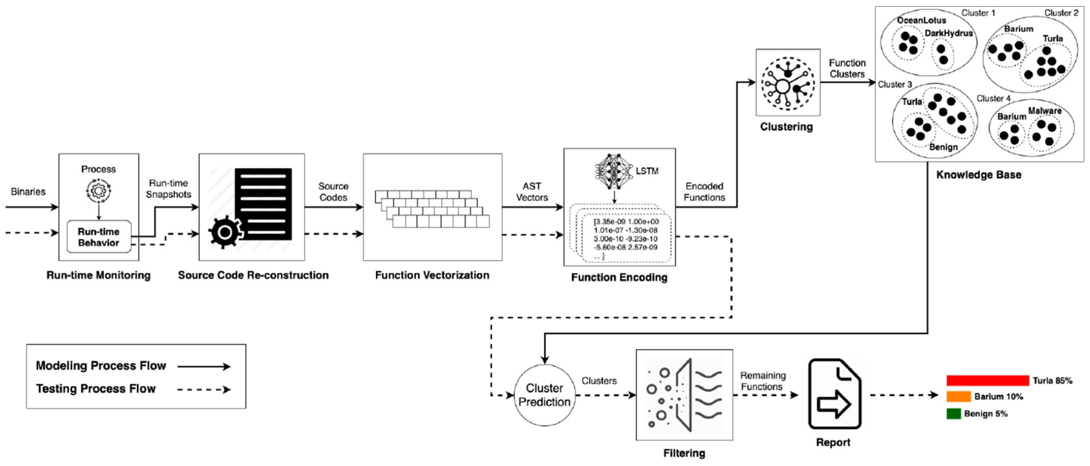


Fig. 1. Overall architecture of the proposed system and steps of modeling (i.e., solid arrows) and testing (i.e., dashed arrows) phases.

2.1 Decompilation

The first step in our pipeline involves binary decompilation to generate the unpacked version of the given payload. To this end, our approach relies on existing dynamic analysis techniques where the given binary is executed in an advanced sandbox developed by Lastline [39], to load and unpack the payload and record the run-time behavior. The sandbox acts as a universal unpacker by running the sample inside a guest operating system.

The dynamic analysis engine produces multiple process dumps (or snapshots) at critical points during different stages of the analysis. An analysis stage is known to be critical if one of these conditions are met: 1) when sensitive APIs are executed, causing a new process creation (e.g., CreateProcess), a new file creation (e.g., CreateFile) or privilege escalation (e.g., AdjustTokenPrivileges), 2) when execution happens outside of the original PE image, and 3) when the original PE image changes. The process dumps contain run-time information about loaded code and memory blocks of the binary under analysis. These snapshots, taken at different stages of the analysis, are then saved under a single analysis subject when the engine detects a suspicious behavior (e.g., a sensitive API call from an untrusted memory region). We parse these snapshots and run a post-process analysis to reconstruct the program's source code in C/C++. To this end, we developed a plugin for Ghidra [24] to map the memory regions from snapshots into Ghidra's virtual memory space and decompile the code found in memory.

2.2 Func2vec Encoding

The second step in the analysis pipeline is designed to perform code similarity testing across malicious payloads. Our analysis on existing similarity testing techniques on source code [4,5,18,35,37,42,63,71] revealed that these approaches are too sparse to such an adversarial landscape where malicious operators have significant freedom to utilize evasive techniques and bypass contemporary similarity testing mechanisms.

Our function to vector encoding mechanism relies on Siamese neural networks [9]. Prior research and our empirical preliminary experiments with malware have shown that SSNs are quite effective in a wide range of tasks, most specifically in similarity and metric learning [56,57] and hashing [44]. SSNs consist of two or more sub-network components with identical architectures. The most common type of Siamese networks is the one with two similar sub-networks which takes two inputs simultaneously and computes two representation vectors (encodings) for inputs. It then leverages a distance metric to estimate the similarity between the two vectors [9]. Each sub-network component (#2 in Fig. 2) is a Long Short-Term Memory (LSTM) [29], an artificial Recurrent Neural Network (RNN) architecture, that has shown to be efficient on a variety of tasks such as anomaly detection [21] and forecasting time series data [26]. LSTMs are also capable of learning long-term dependencies which is not quite possible in other types of networks due to the vanishing gradient problem [7]. To vectorize functions and feed them as inputs to the Siamese network, we rely on Abstract Syntax Trees (ASTs). In particular, we map each function to a flattened version of its AST node types (#1 in Fig. 2). Finally, to compare the final hidden states of two LSTM layers, we leverage the Manhattan distance metric (#5 in Fig. 2), which turns our Siamese network to be of a MaLSTM [52] variant.

To find similar functions across decompiled samples, we start by extracting n-grams of each function vector (i.e., a sequence of AST nodes). We then hash all extracted functions by applying the Locality Sensitive Hashing (LSH) algorithm [33] to these n-grams. LSH can map similar functions to the same hash code which is referred to as a bucket. At the end of the analysis, buckets that contain only a single function are incorporated as dissimilar functions. We then select all permutations of these functions as dissimilar pairs. We have observed that relying on LSH alone normally introduces false positives, especially in cases where functions are extracted from decompiled codes and they have major differences. For this reason, we leverage this hashing mechanism along with two verification techniques to discard false positives and come up with an ML-based hashing mechanism.

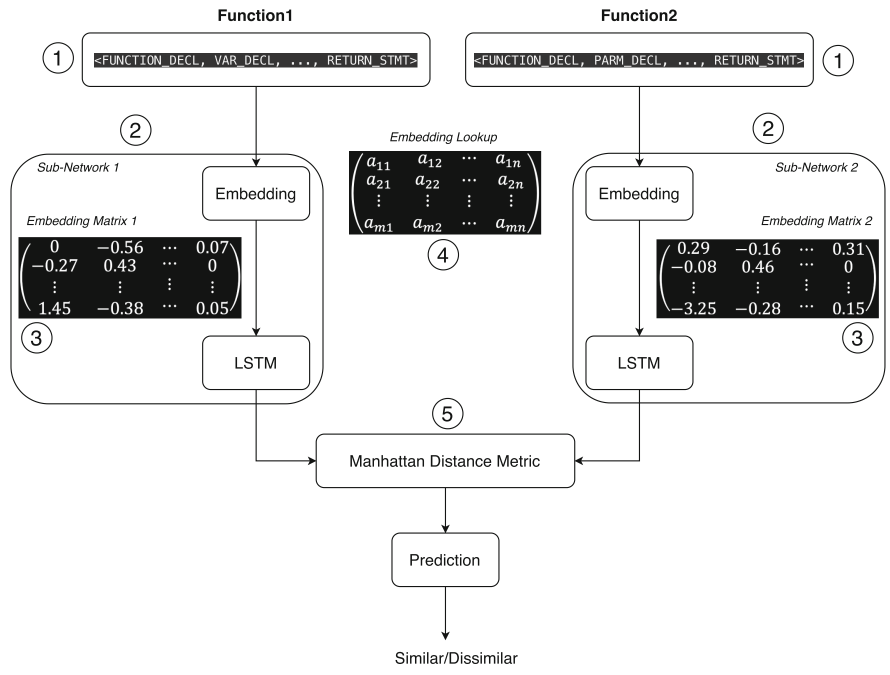


Fig. 2. Overall architecture of Siamese network.

2.3 Encoding Clustering

The primary goal of our analysis pipeline is to provide insights on a previously unknown binary. To this end, we begin by incorporating a trained Siamese network (see Sect. 2.2) to encode functions, and then, leverage an unsupervised learning approach to cluster similar functions from different code samples together. The generated clusters form our knowledge base to locate code reuse in a previously unknown sample and reason about the potential behavior of the sample. That is, the sample is converted to a set of encoded functions which are compared against clusters of hundreds of thousands of encoded functions that correspond to previously detected campaigns or malware families.

For each cluster, we generate a unique dictionary tag by maintaining the occurrence frequency of each function observed in different decompiled code samples. The tag of each cluster reveals the usage or popularity of the clustered functions in malware samples used by one or more campaigns. We have also tags for functions that were used mainly for benign code samples.

Malicious and benign binaries share significant volumes of code samples based on our observations. For instance, statically-linked standard library functions, global variable initialization code, and import resolution code are significantly common in both types. Therefore, a function cannot be simply labeled as malicious if it is observed in malware. Benign code samples could be identified as malicious only because they share several lines of code with malicious binaries. Thus, in this work, we rely on clusters of function encodings and automatically tag new functions as noisy if they are assigned to a mixed cluster that contains functions from both malicious and benign samples. We then discard all these noisy functions before doing any further analysis.

To provide an example of how SCRUTINIZER performs automated code reuse detection on an unknown sample, we extracted 534 functions with 45,902 LoC after running a sample in the sandbox and reconstructing its source code. The system then generated the corresponding encoded functions using the func2vec method for similarity testing via a trained SSN and a knowledge base of approximately 1.7M function encodings from 44K benign and malicious payloads. The system discarded 434 (81%) functions, which was equal to 36,957 LoC, and identified 100 (29%) functions in previously seen malware binaries, most of which were assigned to clusters with *Turla* tag – A Russian-based threat group that has infected victims in over 45 countries. After further investigation, we discovered that this sample was a Trojan package that was suspected by computer security researchers and Western intelligence officers to be the product of a Russian government agency of the same name.

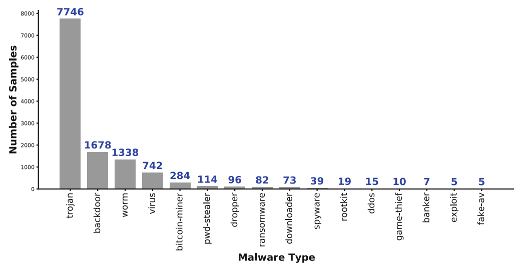
3 Evaluation

In this section, we evaluate our system and discuss the results. In particular, we will explain our experimental setting, the encoding mechanism used to cluster similar functions. We will also provide details on the real-world deployment of SCRUTINIZER and our results in malware campaign analysis.

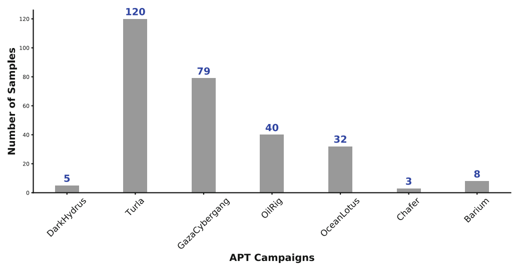
3.1 Experimental Setting and Dataset

SCRUTINIZER is implemented in Python. We have leveraged a library for parallel computing, called Dask [15], to improve the execution speed of our scripts. The experiments are conducted on a 2.2 GHz Intel Xeon Ubuntu server with 20 CPUs and 276 GB of RAM. We have used two distinct and non-overlapping datasets for the modeling and testing phases (see Table 1). To create a model, we have relied on 31,475 benign and 12,540 malware samples. Benign samples are different versions of Windows DLL files crawled from a website [17] in around one week. Table 1 shows the average size for each type of binary, and also, the average Lines of Code (LoC) and cyclomatic complexity [45] of their decompiled codes. Malicious samples (shown in Fig. 3) are both regular malware (12,253 samples) and those that have been used in advanced (i.e., APT) attacks (287 samples) according to the APT notes and threat intelligence of MITRE ATT&CK [50], FireEye [22], Kaspersky [36] and ThreatMiner [66][1]. To test our system, we have increased the malware-to-benign ratio and have evaluated its performance on unknown samples (See Sect. 3.4).

[1] We plan to release a labeled dataset of malware binaries that have been used by different APT campaigns that we have access to.



(a) Distribution of malware samples in different types.



(b) Distribution of malware samples in different APT campaigns [65]

Fig. 3. Distributions of malware in our dataset.

Table 1. Overview of the dataset used in this work. The size (in MB) and cyclomatic complexity are given on average per sample. Avg_LOC is the average lines of code per sample and per function.

Phase	Data type	#Samples	Size	Avg_LOC	Complexity
Modeling	Malware [39]	12,540	0.55	106.21	11.05
	Benign [17]	31,475	0.31	35.73	5.80
Total		44,015			
Testing	Malware [39]	500	0.38	95.47	10.21
	Benign [39]	2,500	0.29	33.25	5.76
Total		3,000			

3.2 Function Encoding

This section particularly deals with answering two important questions: 1) How is a model is trained to embed functions?, and 2) How can the system leverage this encoding method to perform similarity testing? To perform training experiments, we first decompile each binary using a Ghidra[2] post-processing script [64] as discussed in Sect. 2.1. We then leverage Clang[3] to map each function to a flattened AST vector. Our choice of Clang is motivated by the fact that it is less sensitive to code artifacts commonly produced by decompilation tools. To make our AST construction more precise, we replace each function or API call (`CALL_EXPR` node in AST) in the vector by the exact name of a function or API call. This allows us to differentiate between calls to functions authored by the coders and APIs.

We then train the MaLSTM network to identify pairs of similar and dissimilar functions by mapping their AST vectors into an embedding space. We relied on the MinHashing implementation of Locality Sensitive Hashing (LSH) [33], and hashed functions based on the n-grams of their AST vectors. The parameters of fuzzy hashing were chosen very loosely so that we could come up with as many

[2] Version 9.1.2 with SHA-256: ebe3fa...ecac61.

[3] Version 10.0.0: https://releases.llvm.org/10.0.0/tools/clang.

similar functions as possible with different lengths. We followed two major steps to discarded functions that were mistakenly identified as similar (i.e., false positives). First, we leveraged a post-processing script [64] to automatically discard similar functions that do not share a common prototype (i.e., function name and parameters) and output. Second, we hashed a subset of functions several times to manually inspect and verify the integrity of results (i.e., created set of similar and dissimilar functions). In total, we collected 1,105,000 pairs for similar and dissimilar functions from both malicious and benign samples.

The accuracy of the trained network is evaluated via non-stratified 5-fold cross-validation. In particular, the dataset was first split into 5 folds of approximately equal size. Next, in each of the 5 iterations, we trained our system on 4 folds, and then, evaluated it on the remaining fold. The average, median and standard deviation of prediction errors (shown in Table 2) are 0.082, 0.097, 0.056, and 0.061 for malware and benign function vectors respectively that show our system works well. Moreover, we have randomly selected 1,000 function encodings and have manually checked the integrity of the results. Our manual inspection confirms the integrity and accuracy of the encoding mechanism and that similar AST nodes are properly placed close to each other in the latent space.

Table 2. Prediction error statistics after 5-fold cross-validation using only malware, only benign vectors, and a combination of both.

Type	Mean	Standard deviation	Median
Malware	0.082	0.097	0.031
Benign	0.056	0.061	0.004
Both	0.058	0.071	0.017

We also leveraged cross-validation to confirm the optimal value of embedding dimension and to tune the network's hyper-parameters, and in particular, the number of hidden layers. Due to time limit and the number of samples, we evaluated the performance of our system with 5 different dimensions and 3 networks with different numbers of hidden layers. Our results confirmed that an embedding size of 128 yields to higher F1 scores for all three settings.

3.3 Cluster Analysis

Equipped with a precise function-level encoding mechanism and similarity testing, we now hash each function to an encoding of a particular dimension (i.e., Func2vec encoding). To achieve this goal, we process all samples in our dataset and extract their functions and their AST vectors initially. We then generate the encodings of the functions using one of the sub-networks (i.e., LSTM) of our Siamese network (i.e., MaLSTM), cluster them into groups, and use these clusters to discard noisy functions and keep the ones that had been seen mainly in malware samples.

We leveraged the HDBSCAN clustering algorithm [13,46] to group function encodings in different clusters due to its superior performance over other algorithms for big data [59]. This algorithm is a variant of a popular density-based clustering algorithm, known as DBSCAN [20]. It requires no parameter tuning, runs faster and consumes less memory compared to the regular DBSCAN algorithm due to the way it has been implemented. To speed up the clustering process, we have reduced the dimension of our function encodings from 128 to 8 using Principal Component Analysis (PCA). Also, we experimentally tested and approved that reducing the dimension of the data would not significantly impact the clustering result, and most specifically, the number of clusters.

Our system could find a total number of 1,734,992 clusters based on the above parameters and clustering algorithm. Each cluster contains functions whose encodings resemble each other. From this number, 91% of clusters are completely benign, 3.2% are completely malicious, and 5.88% of clusters are mixed. The biggest cluster in our dataset has 14,406 similar function encodings, while the average size of clusters is around 5.

After clustering similar functions together, we generate a dictionary tag for each cluster based on the occurrence frequency of the observed functions in decompiled code samples. The occurrence frequency of a function is shown in Eq. 1. We maintain the occurrence frequency of each function during the modeling phase, and eventually, generate a dictionary tag of all the possible use cases of the function. For instance, the function that we described in Sect. 2.3 was automatically assigned to a cluster that had the following cluster tag:

$$tag = [benign = 12.5, Turla = 87.5] \tag{1}$$

The cluster tags created during clustering are used later for filtering and code reuse detection. To show the performance of SCRUTINIZER in these two tasks, we have applied it on real-world data. Specifically, we first show the efficiency of the filtering process, and then, we discuss how it can improve the accuracy of a malware detection system. Finally, we show how the proposed system could be leveraged to identify code reuse among malware samples that have been used by APT campaigns.

3.4 Real-World Deployment

We received 3,000 previously unknown samples collected by a well-known anti-malware company in the summer of 2020 to test the effectiveness of SCRUTINIZER. Note that based on our empirical analysis, each sample contained on average of 485 functions. This makes code similarity testing a significantly expensive process. More importantly, malware and benign samples share large volumes of standard code. This overlap could potentially introduce many false positives in malware detectors. Thus, we first filter noisy functions that are common in both malware and benign samples, and then, rely on the remaining functions of each sample to perform similarity analysis. In the following, we provide more details on each step as well as the summary of our experiments.

Filtering. The filtering process is applied by relying on cluster tags that are created during the clustering process (see Sect. 3.3). After the cluster assignment, the system discards functions if the cluster tag indicates that all or the majority of the functions (enforced by δ) in the assigned cluster are observed in the benign samples. It is worth mentioning that δ is customizable. Therefore, a higher value would only discard functions that were seen in benign clusters. We set the δ value to 0.5 to discard functions because this threshold value left us with more functions in malicious clusters on average. The summary of the filtering experiment is quite promising (See Fig. 4). The analysis shows that the samples in the previously unknown dataset have a median of 199 functions where the system was able to filter a median of 126 functions (63%) from those samples. This was equal to removing approximately 11,476 (56%) lines of code from the given code.

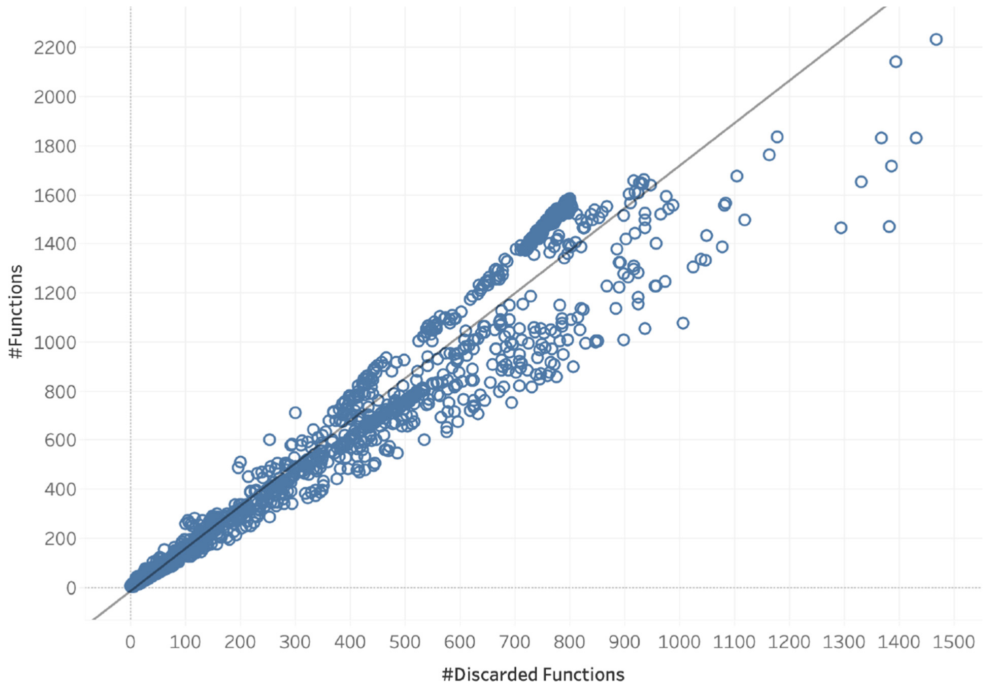


Fig. 4. The result of the filtering phase on the unknown sample set. Our analysis suggests that the filtering mechanism works well in practice by filtering a median of 126 functions of the input samples.

The filtering process had a significant impact on the performance of our system. To confirm this, we deployed SCRUTINIZER to detect unknown malware. Here, the system was tuned to label any incoming unknown sample as malware when malicious functions outnumbered benign ones, and the similarity of the sample to any previously known samples was more than a threshold, ranging from 0 to 100%. The experimental results show that the true positive rate increased from 82% to 92% after filtering the functions that were common in both malware and benign samples. At the same time, the false positive rate decreased from 10% to around 1.2%. In other words, as shown in Fig. 5, the Area Under the Curve (AUC) score, ranging from 0 (worst classifier) to 1 (perfect classifier), improved from 0.88 to 0.95 when the filtering was applied. In Sect. 3.4, we explain how the output of this section helped us to conduct the malware similarity analysis on this dataset.

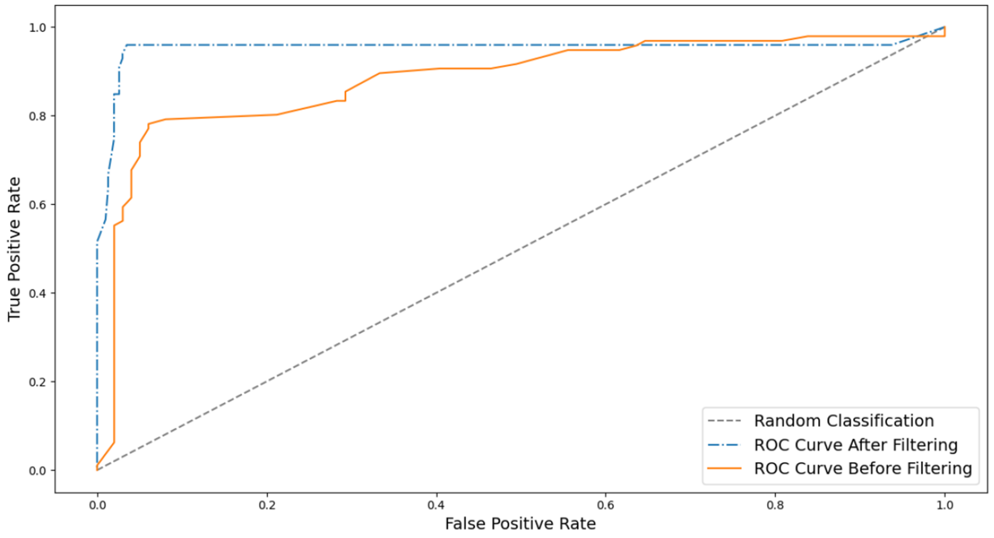


Fig. 5. Classification performance before and after filtering process. The results suggest that the applied filtering mechanism improves the TPR by 10% and decreases the FPR by 8.8%.

Code Reuse Analysis on APT Campaigns. One of our motivations behind developing SCRUTINIZER was to enhance contemporary techniques to perform malware similarity analysis on unknown samples. In this experiment, we tested SCRUTINIZER by using all the 3,000 unknown samples we discussed in Sect. 3.4. We followed the steps we described in Sect. 2 to perform this experiment. In this Section, we discuss two case studies: 1) a scenario where chunks of codes were identified by our system to be shared by malware samples that had been used by one single APT campaign, and 2) a scenario were code was shared and re-used by multiple APT campaigns.

Intra-campaign Code Reuse Analysis. In the first case study, we discuss how SCRUTINIZER can incorporate function-level similarity testing and automatically reason about code reuse in a given sample. In our analysis, we found a case[4] which was identified to have function-level similarities with malware specimens attributed to OceanLotus Vietnamese APT campaign.

The very initial malicious operation of this cyber-theft and espionage APT campaign dates back to 2014, where a European corporation was compromised [55]. Since then, this campaign has been continuously active with the maximum number of operations in 2018. The most recent and known activity of

[4] MD5: fcd7227891271a65b729a27de962c0cb.

this campaign has been focused on targeting the Wuhan government and Chinese ministry of emergency management to steal COVID-19 related findings via sending phishing emails [54].

Our automated system found 377 different functions from this sample with 18,125 lines of code. From this amount, SCRUTINIZER could discard 247 noisy functions which were up to 8,720 lines of code - i.e., around 48% of the original decompiled code. Next, the remaining functions were assigned to 109 different clusters, where in the best case, 4 functions were grouped in one single cluster. Our results show that this sample contains 105 different functions similar to those functions that were seen in malicious specimens of the OceanLotus APT campaign that was observed in the modeling phase. We cross-checked these results with major anti-malware service providers. The sample was recognized as Ocean Buffalo by CrowdStrike [53], or APT32 by FireEye [22].

Inter-campaign Code Reuse Analysis. In the second case study, we identified a sample[5] that had function-level similarities with malware specimens attributed to the Barium and Turla APT campaigns – two well-known Chinese and Russian state-sponsored campaigns that have cyber-espionage and information theft motives [65].

The Barium campaign commonly begins its attacks by gathering information of potential victims through social engineering techniques, especially via online social media. Multimedia and online game companies were attacked more frequently by this campaign. The Turla campaign, also known as Waterbug by Symantec [70], has targeted a wider range of users and sectors in over 45 different countries. The threat intelligence report also shows that this campaign is one of the most sophisticated APT campaigns that relies on a wide range of tools to deliver its attacks [65].

Our system found 971 functions from this sample with 74,309 lines of code overall. Also, it discarded 767 noisy functions which corresponded to 59,976 lines of code (≈81% of the original decompiled code). Next, the remaining 204 functions were assigned to 163 different clusters. Experimental results show that this sample has 22 functions that are syntactically similar to those of malware samples attributed to both the Barium and Turla campaigns. After further investigation, we discovered that this sample contains an open-source hacking code known as Mimikatz [49] that has not only been used by these two campaigns, but also by a number of other advanced malware campaigns [65].

Summary. Table 3 shows the list of all binaries that our system reported to have similarities with specimens that were known from each advanced malware (i.e., APT) campaign that existed in our knowledge base. We could verify the veracity of our results by consulting online AV scanners and threat intelligence reports. Some of these binary hashes were already reported as IoCs of known APT campaigns. Others were cross-checked by retrieving the name of the malicious operation or malware family and consulting the recent APT threat ency-

[5] MD5: 276c28759d06e09a28524fffc2812580.

clopedia [65]. The results included in Table 3 confirm that our tool can reason about code reuse in unknown samples by relying on function-level similarities.

Specifically, a similarity score presented for each sample by our tool shows how many functions an unknown sample has in common with already known malware instances. Moreover, this score represents how frequently a function is observed in each campaign. A low similarity score for an unknown malicious binary shows that it has less shared functions with known malware specimens of a specific campaign. On the other hand, a high similarity score indicates that the unknown binary has more functions in common with known malware samples that have been used by a malware campaign. Thus, the proposed approach to perform similarity testing can be used in different ways depending on the goal of the analysis. That is, if a human analyst aims at locating new and emerging trends in malware samples, a lower score is more preferable. If, however, the analyst is looking for code reuse in unknown samples, a higher score is an indicator of more overlaps.

Table 3. Campaign analysis result for a subset of samples that we could manually verify using online threat reports and AV scanners.

MD5	#Functions	Discarded functions (%)	Assigned campaign: similarity (%)	Real campaign
22d01fa2725ad7a83948f399144563f9	763	81.9	Turla: 58.0	Turla [69]
0d67422ba42d4a548e807b0298e372c7	225	55.1	GazaCybergang: 73.9	GazaCybergang [51]
655f56f880655198962ca8dd746431e8	188	66.5	GazaCybergang: 64.0	GazaCybergang [51]
ff8d92dfbcda572ef97c142017eec658	144	70.1	Barium: 38.5	Barium [65,69]
c11dd805de683822bf4922aecb9bfef5	220	65.9	Barium: 38.4	Barium [65,69]
aae531a922d9cca9ddca3d98be09f9df	558	61.6	OilRig: 43.7	OilRig [65,69]
6a7bff614a1c2fd2901a5bd1d878be59	588	59.0	OilRig: 40.6	OilRig [65,69]
a921aa35deedf09fabee767824fd8f7e	44	68.2	GazaCybergang: 41.5	GazaCybergang [65,69]
0e441602449856e57d1105496023f458	73	61.6	Turla: 35.3	Turla [69]
7f05d410dc0d1b0e7a3fcc6cdda7a2ff	220	65.9	Barium: 38.4	Barium [65,69]
557ff68798c71652db8a85596a4bab72	144	70.1	Barium: 38.5	Barium [65,69]
b0877494d36fab1f9f4219c3defbfb19	144	70.1	Barium: 38.5	Barium [65,69]

4 Discussion

In this section, we discuss the accuracy, robustness and potential deployment costs of our approach.

Accuracy. The very first step of our pipeline involves mapping memory regions from snapshots taken during dynamic analysis into Ghidra's virtual memory space and use this tool to decompile the code found in memory. Thus, any issues that may happen during this step could impact the accuracy of our tool. In addition, like any other machine learning approach, the accuracy of SCRUTINIZER depends on the quantity of the data that is used to build the detection models. In particular, a prerequisite in training a Siamese network is the availability of a large number of function pairs from different binaries. This process requires extracting the corresponding syntactical features after constructing the decompiled version of the binary. However, collecting a large volume of relevant data is a non-trivial task since binary decompilation is an error-prone process on malware binaries, and it could contribute to lower accuracy. Furthermore, our experiments show that the AST construction method we used in this project can also produce false positives. In particular, we empirically found that in specific cases, ASTs produced by Clang showed similarities that were false positives. As a result, there might be noise in the training data and this might affect the function encoding process. To cope with this problem, we leveraged a post-processing mechanism to carefully inspect the result of the encoding process, and we performed hyper-parameter tunings to prefilter false positives. We note that in practice, the post-processing operation needs to be constantly visited and updated to keep the false positive rate in a manageable range.

Analysis Costs and Potential Bottlenecks. Our analysis of the performance overhead revealed three sources of potential bottlenecks in the process. Recall that our analysis relies on dynamic analysis which is inherently an expensive operation in code analysis. Common evasion techniques could even increase this cost by inserting stalling code or logic bombs and make the dynamic analysis significantly less effective. The analysis framework we used in this project allowed us to record the actual malicious behavior even with the presence of logic and time bombs. However, this is a best-effort approach, and requires constant improvement to maintain effectiveness which outside of the scope of this paper. The second expensive task in this process is related to constructing the training data for the Siamese network. This phase requires performing pair-wise similarity testing across all the decompiled code instances. The similarity testing on all functions in 1,000 decompiled code samples required around 4 h. However, note that this process needs to be performed once in order to build the training dataset, and subsequent incremental updates would not introduce signification overhead. The third potential computational cost is related to the training process where several parameters such as hyper-parameters and epochs need to be considered for constructing the model. In this work, each of the sub-networks were trained in 5 epochs in parallel, and the training process on around 1M pairs of functions took approximately 36 min. We observed that the processing time can dramatically increase when we incorporated larger pairs of functions. However, the results do suggest that a larger number of pairs (e.g., 10M) did not necessarily contribute to significantly more accurate results.

Practical Deployment. We posit that a number of useful activities can be performed with our tool such as approximating the prevalence of emerging advanced malware threats, filtering previously known binaries, or identifying the adoption of new attack techniques across different campaigns. The proposed approach to perform code reuse analysis would allow human analysts to locate emerging trends by looking for code samples with lower similarity scores. For instance, we automatically identified several variants of OceanLotus campaign by simply looking for code samples with similarity score less than 15%. Furthermore, by applying a higher similarity score, it is possible to perform scalable unsupervised learning and group more relevant code based on their function-level similarity.

We empirically showed that this approach works well in practice by analyzing 3,000 previously unknown samples (see Sect. 3).

5 Related Work

Binary Code Similarity Detection. Binary code similarity detection has been applied to many applications, including code plagiarism detection [43,62], malware family and lineage analysis [6,34], and vulnerability analysis [10] to name a few.

State-of-the-art BCSD solutions heavily rely on a specific syntactic feature of binary code extracted via static analysis such as control flow graphs (CFGs) [8,19,23,60]. The majority of these solutions have low accuracy, especially when it comes to advanced malware that leverages several anti-analysis techniques to circumvent the static analysis. On the other hand, few solutions are proposed that are more resilient against anti-analysis techniques such as obfuscation and have considered semantics of binary codes to identify possible similarities. Nevertheless, they have not been applied to malicious binaries. Contrary to these solutions, SCRUTINIZER relies on syntactic features that are extracted from decompiled memory snapshots taken during dynamic analysis.

Authorship Attribution. Previous studies have also explored different ways through which users can be deanonymized based on their coding styles. Two broad categories of methods have been investigated in academia to address this problem that work either at the binary-level [11,47,61] or at the source code level [1,12,16]. Binary-level methods, while fast and useful, work under the assumption that a toolchain provenance is used to generate the binary, including a specific compiler, operating system and source language. In contrast, code-level methods are more flexible, specifically because coding styles of authors at the source code level are not lost during compilation. Recent works have relied on coding style features and have leveraged machine learning to develop more robust mechanisms for code authorship attribution. For example, Abuhamad et al. [1] have proposed a system that attributes code at a large scale effectively using deep neural networks. Caliskan-Islam et al. have relied on random forests and syntactic features from ASTs to de-anonymize code authors based on their coding styles [2,12]. While these approaches are effective in attributing code to specific authors, they are less effective when dealing with adversarial code that has been decompiled and where coding style features such as the naming conventions for variables have been lost. In comparison to existing authorship attribution work, our work fills the gap and addresses the code similarity detection problem in code that is only available as binary, and where source code needs to be extracted.

Malware Clustering. Automated malware clustering has received well-deserved attention in the past as it helps to identify the type and severity of the threat that each malware specimen constitutes. Also, it is useful in tracing new trends in malware samples and creating detection signatures and removal procedures. These approaches can be categorized into three main groups depending on their feature extraction strategy. The first category of approaches rely on features that are extracted statically before the binary is executed. For example, MutantX-S [32] is an automated tool that relies on code instruction sentences to cluster malware samples. The second category of approaches leverage features that are extracted at run-time, normally by running the sample in an emulated environment. As an example, Bayer et al. [6] run each binary in a sandbox and cluster malware specimens based on their behavioral profiles and how they interact with the operating system. Finally, the third category makes use of features that are extracted before and after the execution time. An example includes DUET [31], a tool that relies on both static and dynamic features to cluster malware binaries. Note that these tools are primarily interested in automatically identifying to what family an unknown, obfuscated (or encrypted) sample belongs to. They are not focused on coding patterns to be able to determine campaign similarity or potential attribution.

6 Conclusion

In this paper, we presented an automated system for malicious code similarity identification and campaign attribution. The system decompiles binaries of both malicious and benign applications and encodes their functions using Siamese networks. It then clusters function encodings into different groups and leverages cluster tags created during clustering to facilitate the analysis of new samples that anti-malware companies receive every day. We deployed SCRUTINIZER in a real-world setting and it proved to be useful in both function filtering (i.e., reverse engineering) and code reuse analysis on APT campaigns. Using this system, we were able to identify 12 samples that were previously unknown to us and that we were able to correctly assign to well-known APT campaigns.

Acknowledgement. This work was partially-supported by National Science Foundation (NSF) under grant CNS-1703454, and the Office of Naval Research (ONR) under the "In Situ Malware" project. This work was also partially-supported by Secure Business Austria.

References

1. Abuhamad, M., AbuHmed, T., Mohaisen, A., Nyang, D.: Large-scale and language-oblivious code authorship identification. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 101–114 (2018)
2. Afroz, S., Islam, A.C., Stolerman, A., Greenstadt, R., McCoy, D.: Doppelgänger finder: taking stylometry to the underground. In: 2014 IEEE Symposium on Security and Privacy, pp. 212–226. IEEE (2014)
3. APT trends report Q1 2020 (2020). https://securelist.com/apt-trends-report-q1-2020/96826/. Accessed 05 July 2020
4. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering, pp. 86–95. IEEE (1995)

5. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS/SPL REG: program transformations for practical scalable software evolution. In: Proceedings of 26th International Conference on Software Engineering, pp. 625–634. IEEE (2004)
6. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: NDSS, vol. 9, pp. 8–11. Citeseer (2009)
7. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
8. Bindiff: a comparison tool for binary files. https://www.zynamics.com/bindiff.html (2020). Accessed 05 May 2020
9. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R.: Signature verification using a "siamese" time delay neural network. In: Advances in Neural Information Processing Systems, pp. 737–744 (1994)
10. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: 2008 IEEE Symposium on Security and Privacy (SP 2008), pp. 143–157. IEEE (2008)
11. Caliskan, A., et al.: When coding style survives compilation: de-anonymizing programmers from executable binaries. arXiv preprint arXiv:1512.08546 (2015)
12. Caliskan-Islam, A., et al.: De-anonymizing programmers via code stylometry. In: 24th USENIX Security Symposium (USENIX Security 2015), pp. 255–270 (2015)
13. Campello, R.J.G.B., Moulavi, D., Sander, J.: Density-based clustering based on hierarchical density estimates. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds.) PAKDD 2013. LNCS (LNAI), vol. 7819, pp. 160–172. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37456-2_14
14. APT Groups Target Healthcare and Essential Services. https://us-cert.cisa.gov/ncas/alerts/AA20126A (2020). Accessed 05 May 2020
15. Dask: A flexible library for parallel computing in python. https://docs.dask.org (2018). Accessed 05 May 2020
16. Dauber, E., et al.: Git blame who?: stylistic authorship attribution of small, incomplete source code fragments. Proc. Privacy Enhanc. Technol. **2019**(3), 389–408 (2019)
17. DLL Files. https://www.dll-files.com (2020). Accessed 14 Mar 2020
18. Ducau, F.N., Rudd, E.M., Heppner, T.M., Long, A., Berlin, K.: SMART: semantic malware attribute relevance tagging. CoRR abs/1905.06262 (2019). http://arxiv.org/abs/1905.06262
19. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: efficient cross-architecture identification of bugs in binary code. In: NDSS (2016)
20. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. KDD **96**, 226–231 (1996)
21. Feng, C., Li, T., Chana, D.: Multi-level anomaly detection in industrial control systems via package signatures and LSTM networks. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 261–272. IEEE (2017)
22. Advanced Persistent Threat Groups. https://www.fireeye.com/current-threats/apt-groups.html (2020). Accessed 14 Mar 2020
23. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88625-9_16
24. Ghidra: A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate. https://ghidra-sre.org (2020). Accessed 14 Mar 2020

25. Graziano, M., et al.: Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence. In: 24th USENIX Security Symposium (USENIX Security 2015), pp. 1057–1072 (2015)
26. Guo, T., Xu, Z., Yao, X., Chen, H., Aberer, K., Funaya, K.: Robust online time series prediction with recurrent neural networks. In: 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), pp. 816–825. IEEE (2016)
27. Han, X., Pasquier, T., Bates, A., Mickens, J., Seltzer, M.: UNICORN: runtime provenance-based detector for advanced persistent threats. In: NDSS (2020)
28. Hardy, S., et al.: Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 527–541 (2014)
29. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
30. Hossain, M.N., et al.: SLEUTH: real-time attack scenario reconstruction from COTS audit data. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 487–504 (2017)
31. Hu, X., Shin, K.G.: Duet: integration of dynamic and static analyses for malware clustering with cluster ensembles. In: Proceedings of the 29th Annual Computer Security Applications Conference, pp. 79–88 (2013)
32. Hu, X., Shin, K.G., Bhatkar, S., Griffin, K.: Mutantx-s: scalable malware clustering based on static features. In: 2013 USENIX Annual Technical Conference (USENIX ATC 2013), pp. 187–198 (2013)
33. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613 (1998)
34. Jang, J., Woo, M., Brumley, D.: Towards automatic software lineage inference. In: 22nd USENIX Security Symposium (USENIX Security 2013), pp. 81–96 (2013)
35. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 96–105. IEEE (2007)
36. Targeted Cyberattacks Logbook. https://apt.securelist.com/#!/threats/ (2018). Accessed 14 Mar 2020
37. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_3
38. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. **3**, 91–97 (2006)
39. Lastline. https://www.lastline.com (2021). Accessed 04 May 2021
40. Le Blond, S., Uritesc, A., Gilbert, C., Chua, Z.L., Saxena, P., Kirda, E.: A look at targeted attacks through the lense of an NGO. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 543–558 (2014)
41. Li, Y., et al.: Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 2015) (2015)
42. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-miner: a tool for finding copy-paste and related bugs in operating system code. OSdi **4**, 289–302 (2004)
43. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 389–400 (2014)

44. Masci, J., Bronstein, M.M., Bronstein, A.M., Schmidhuber, J.: Multimodal similarity-preserving hashing. IEEE Trans. Pattern Anal. Mach. Intell. **36**(4), 824–830 (2013)
45. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)
46. McInnes, L., Healy, J.: Accelerated hierarchical density based clustering. In: 2017 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 33–42. IEEE (2017)
47. Meng, X., Miller, B.P., Jun, K.-S.: Identifying multiple authors in a binary program. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 286–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_16
48. Milajerdi, S.M., Gjomemo, R., Eshete, B., Sekar, R., Venkatakrishnan, V.: Holmes: real-time apt detection through correlation of suspicious information flows. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1137–1152. IEEE (2019)
49. Mimikatz: an open-source application for veiwing and saving authentication credentials (2014). https://github.com/gentilkiwi/mimikatz. Accessed 05 May 2020
50. MITRE ATT&CK: a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations (2020). https://attack.mitre.org/. Accessed 14 Mar 2020
51. Moonlight - Targeted attacks in the Middle East (2016). https://tinyurl.com/45m3jtx8. Accessed 05 July 2020
52. Mueller, J., Thyagarajan, A.: Siamese recurrent architectures for learning sentence similarity. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)
53. Meet the threat actors: List of APTs and adversary groups (2019). https://www.crowdstrike.com/blog/meet-the-adversaries/. Accessed 05 May 2020
54. Vietnamese Threat Actors APT32 Targeting Wuhan Government and Chinese Ministry of Emergency Management in Latest Example of COVID-19 Related Espionage (2020). https://tinyurl.com/7whx7ecr. Accessed 05 May 2020
55. Cyber espionage is alive and well: Apt32 and the threat to global corporations (2017). https://tinyurl.com/54eact6v. Accessed 05 May 2020
56. Oh Song, H., Jegelka, S., Rathod, V., Murphy, K.: Deep metric learning via facility location. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5382–5390 (2017)
57. Oh Song, H., Xiang, Y., Jegelka, S., Savarese, S.: Deep metric learning via lifted structured feature embedding. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4004–4012 (2016)
58. Oliver, J., Cheng, C., Chen, Y.: TLSH-a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop, pp. 7–13. IEEE (2013)
59. Benchmarking performance and scaling of python clustering algorithms (2020). https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html. Accessed 05 May 2020
60. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy, pp. 709–724. IEEE (2015)
61. Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? Identifying the authors of program binaries. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 172–189. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23822-2_10
62. Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., Su, Z.: Detecting code clones in binary executables. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 117–128 (2009)

63. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85 (2003)
64. Scrutinizer: Detecting code reuse in malware via decompilation and machine learning (2021). https://github.com/OMirzaei/SCRUTINIZER. Accessed 04 May 2021
65. THREAT GROUP CARDS: A threat actor encyclopedia (2019). https://tinyurl.com/bb8mt23k. Accessed 05 Oct 2019
66. ThreatMiner: Data Mining for Threat Intelligence (2020). https://www.threatminer.org/index.php. Accessed 14 Mar 2020
67. Upchurch, J., Zhou, X.: Variant: a malware similarity testing framework. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 31–39. IEEE (2015)
68. Verizon's 2020 data breach investigations report (2020). https://tinyurl.com/56m7m9ym. Accessed 05 May 2020
69. VirusTotal (2020). https://www.virustotal.com/gui/home/search. Accessed 05 June 2020
70. Waterbug: Espionage Group Rolls Out Brand-New Toolset in Attacks Against Governments (2020). https://tinyurl.com/92s76xdn. Accessed 05 May 2020
71. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: 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, pp. 363–376 (2017)