JARV1S: Phenotype Clone Search for Rapid Zero-Day Malware Triage and Functional Decomposition for Cyber Threat Intelligence

Christopher Molloy
Research Assistant
School of Computing
Queen's University
Kingston, ON, Canada
chris.molloy@queensu.ca

Philippe Charland
Defence Scientist
Mission Critical Cyber Security Section
Defence Research and Development Canada
Quebec, QC, Canada
philippe.charland@drdc-rddc.gc.ca

Steven H. H. Ding
Assistant Professor
School of Computing
Queen's University
Kingston, ON, Canada
ding@cs.queensu.ca

Benjamin C. M. Fung
Professor
School of Information Studies
McGill University
Montreal, QC, Canada
ben.fung@mcgill.ca

Abstract: Cyber Threat Intelligence (CTI) has become a critical component of organizations defense against the steady surge of cyber attacks. Malware is one of the most challenging problems for CTI, due to its prevalence, the massive number of variants, and the constantly changing threat actor behaviors. Out of 15 million pieces of new malware collected by Panda Security, less than 1% occurred more than once. The signature-based representation of patterns and knowledge of past legacy systems can no longer be generalized for future attacks. Machine learning-based solutions can match more variants. However, as a black-box approach, they lack the explainability and maintainability required by incident response teams.

There is thus an urgent need for a data-driven system that can abstract a future-proof, human-friendly, systematic, actionable, and dependable knowledge representation from software artifacts from the past for more effective and insightful malware triage. In this paper, we present the first phenotype-based malware decomposition system for quick malware triage that is effective against malware variants. We define phenotype as directly observable characteristics such as code fragments, constants, functions, and strings. Malware development rarely starts from scratch and there are many reused components and code fragments. Given the target under investigation, we decompose it into known phenotypes that are mapped to known malware families, malware behaviors, and Advanced Persistent Threat (APT) groups. The implemented system provides visualizable phenotypes through an interactive tree map, assisting the cyber analyst to navigate through the decomposition results. We evaluate our system on 200,000 malware samples, 100,000 benign samples, and a malware family that has over 27,284 variants. The results indicate our system is scalable, efficient, and effective against zero-day malware and new variants of known families.

Keywords: *malware analysis, malware triage, static analysis, binary clone search, information retrieval*

1. INTRODUCTION

Code reuse has been a common practice and strategy in software engineering, given the vastly available open source repositories [1]. There are no exceptions for the Advanced Persistent Threat (APT) groups that drive the development of malware. Malware samples are not created from scratch, thanks to the readily

available open source components, code generation tools, and Malware-as-a-Service (MaaS) platforms. Malware samples exhibiting similar high-level behaviors are typically considered as variants of the same malware family. These variants are created with code mutation and code obfuscation techniques to evade detection tools [2]. Malware variants under the same malware family therefore share common code to achieve the same or similar set of behavior action sequences. In recent years, malware variants have exponentially increased in both volume and threat potential. Emotet, a popular ransomware tool with more than 70,000 variants has caused upward of USD 1 million per incident for governments and private institutions[1][2]. According to the AV-TEST Institute, 21.8 million new malware variants that target machines running Microsoft operating systems have been recorded in 2021[3]. Due to the ever-increasing number of malware variants, malware samples are categorized into families based on their code, runtime behavior, and functionality. Matching unknown variants to the correct malware families and recognizing novel malware from the same unknown families is a critical process for a successful malware triage in Cyber Threat Intelligence (CTI). Typical methods for malware analysis include dynamic analysis, signature-based pattern matching, and machine-learning-based detection methods. Dynamic analysis is done by observing the behavior of a malware sample in an isolated virtual environment [3] by executing the malware sample itself. The behavior of the malware is recorded and compared with the behavior database of other malware variants. However, modern malware samples have been equipped with anti-sandboxing techniques that can recognize when they are executed in a virtual environment and avoid executing their malicious payload [4]. Additionally, dynamic analysis is a time-consuming process that can take up to two minutes for the analysis of a single sample [3], which fails the time-sensitivity requirement of a CTI pipeline.

Typical anti-virus software, such as McAfee and Microsoft Defender, use a signature-based approach for malware classification. Signature-based pattern matching is based on the method of extracting a small string of identifiable bytes from unknown malware and comparing it to the signatures of known malware [5]. Although signature-based methods can identify malware apart of known families with a low error rate, obfuscation and packing techniques can be implemented by malware authors to impede the signature extraction process [5]. To address this problem, machine learning solutions have been developed for malware clone search [6, 7, 8, 9]. Although these methods have been shown to work with a high number of variants, they do not provide any explanation for their results and the models constantly need to be re-trained and tuned against new malware families using new code obfuscation and packing techniques. Advanced Persistent Threats (APTs) behind the malware attacks are a constant and evolving threat to organizations and governments. To ensure a strong defence against these constantly changing threats, we must keep moving against the adversaries in defence development.

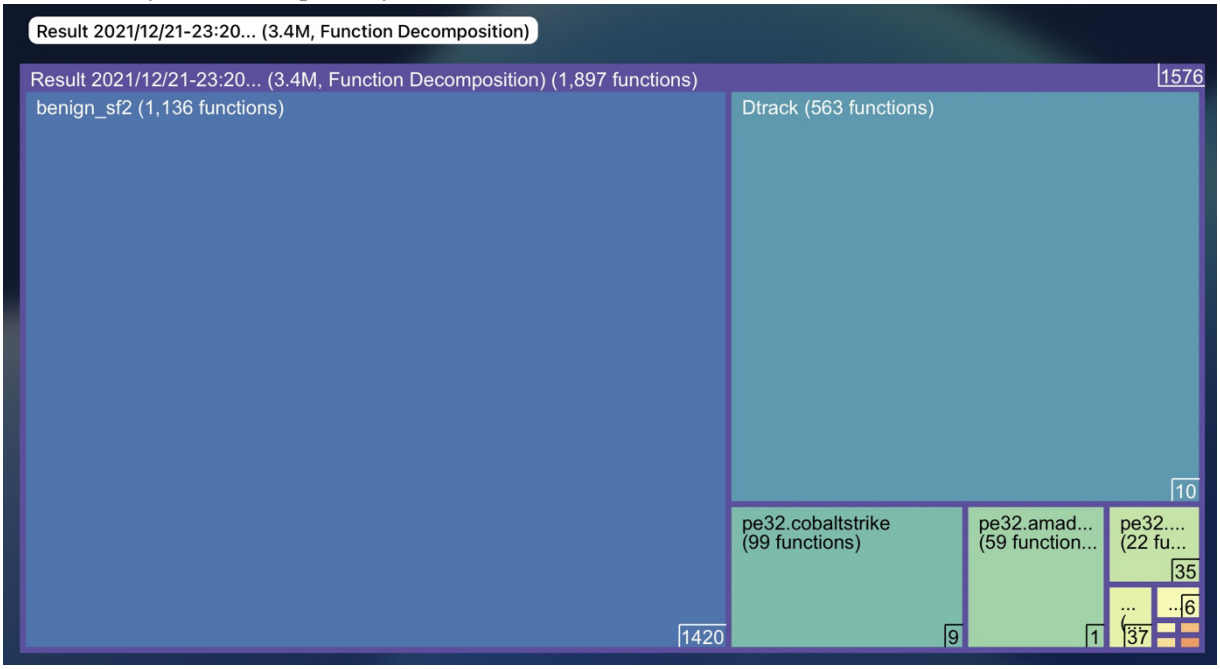
We propose a different approach, namely Phenotype Clone Search for Functional Decomposition. We use phenotypes as the basic elements of observable characteristics extracted from a given malware sample. Examples of phenotypes are code fragments, strings, and constants. They can all be directly observed without sandboxing or emulation. Since a phenotype is defined to be an observable small piece of information, humans can easily interpret its meaning. Malware variants under a given malware family share a similar codebase and therefore, they will share many phenotypes. The organization and

[1] https://www.malwarebytes.com/emotet

[2] https://bazaar.abuse.ch/browse/tag/Emotet/

[3] https://www.av-test.org/en/statistics/malware/

Figure 1: An example of malware functional decomposition. Given a malware sample under analysis (Dtrack family), our system efficiently breaks down the sample into groups of known units of functionality from the repository



presentation of phenotypes may change from sample to sample, but they are the basic elements of a malware executable that cannot easily be hidden. In our proposed approach, we define the basic unit of malware functionality as an assembly function extracted from a malware sample. As a unit of functionality, an assembly function contains a set of observable phenotypes. By matching phenotypes, we can search for cloned units of malware functionality (in this case, the cloned assembly functions) and find the shared functionalities between malware samples. Given a target piece of malware under investigation and a repository of known malware, the objective is to decompose the target into known units of malware functionalities from the repository. Figure 1 shows an example. The target under analysis is a sample of the *Dtrack* malware family. Dtrack is a Remote Administration Tool (RAT) malware that has been found in a nuclear power plant[4]. Our approach decomposes the functions of the target malware into known functions from the repository. We found that around 11,000 of the functions are coming from common code that can be found in benign software. For the rest of the functions, 563 of them can be found in functions from samples of the same family. Additionally, some common malware code that is shared with other families was also found. For zero-day malware samples that do not belong yet to any existing families, we can also study the compositional relationship by searching them against the repository. Our contributions can be summarized as follow:

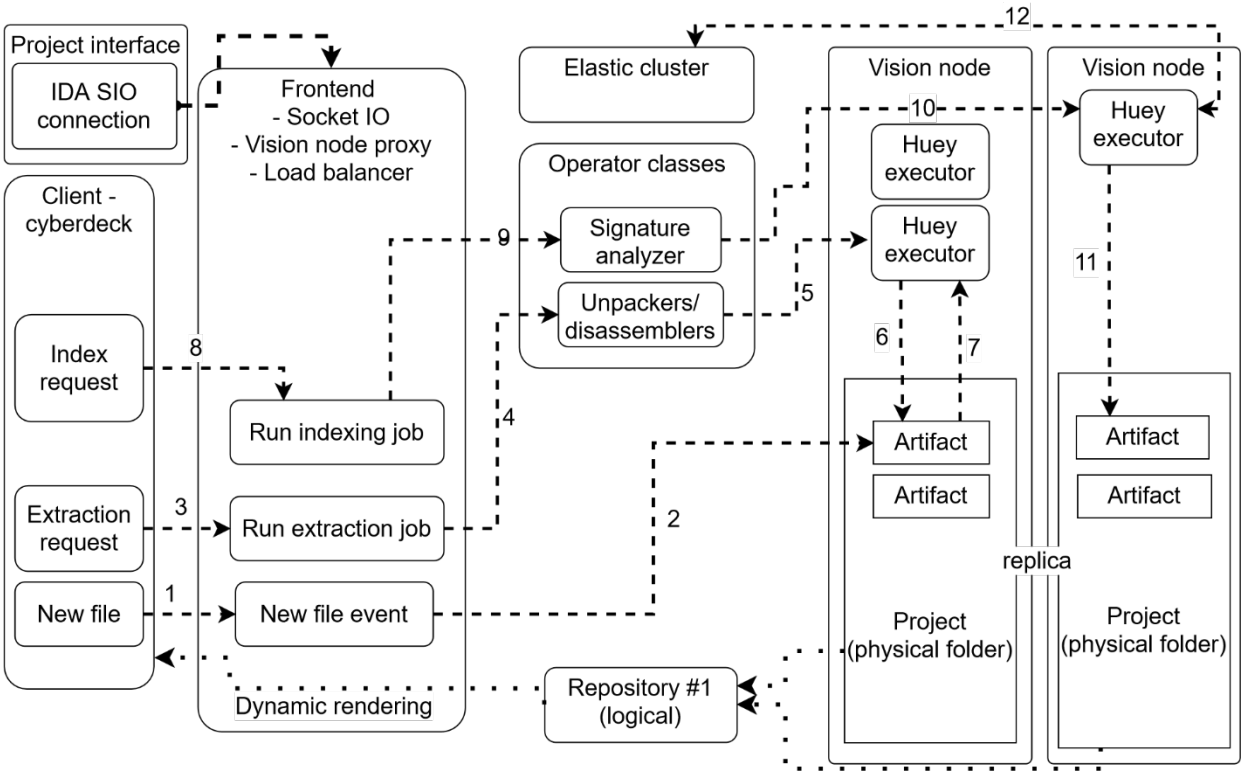
- We propose a novel approach for malware functional decomposition, as a complement over existing dynamic, signature-based, and machine learning-based malware triage solutions.
- We design a new distributed clone search-based analytic framework that enables an efficient, accurate, and robust malware decomposition implementation.

[4] Dtrack: In-depth analysis of APT on a nuclear power plant.

- The proposed solution can decompose a given target malware into known functionalities with interactive visualization, scalable to malware samples of exceptionally large size.

The rest of this paper is organized as follows. Section 2 describes the overall design of the JARV1S system as the host to the proposed malware decomposition method. Section 3 elaborates the clone search method and how we use it to decompose malware functionalities. Section 4 demonstrates the effectiveness of our system. Section 5 discusses the related research in this domain. Finally, Section 6 provides the conclusion.

Figure 2: JARV1S system components and the flow diagram for the malware extraction and indexing process



2. System Design

The proposed JARV1S system itself is an extendable open design platform that aims at hosting scalable and efficient malware analysis algorithms of diverse types and accommodating their special needs. There are several design considerations involved. The first one concerns storage systems and job execution workers. The scale of data to be processed is unlikely to fit into the limited resources one can find in a single server. Additionally, a single server suffers from the problem of single point of failure. There are existing distributed data storage that we can leverage. However, typically distributed storage nodes are in a separate cluster of nodes than the nodes used for job execution. This implies that one will have to maintain at least three separate clusters: storage nodes, execution nodes, and messaging queue nodes. Additionally, one of the typical ways to improve the cluster performance and throughput is data locality: a given job is assigned to a worker, which is co-located with the data in the same node. Separating the computation nodes and storage nodes will completely remove the advantage of data locality. Therefore, we propose and implement our own distributed storage and job execution distributed cluster. Since we know our data models and attributes regarding the information extracted from a malware artifact and they are unlikely to change significantly in the future, the storage system can be further simplified to better

balance the retrieval speed versus the compression ratio. Vision, the storage and job execution cluster of JARV1S, contains three components:

- *Load balancer and Remote Procedure Call (RPC) proxy*. This is the main interface between the Vision cluster and the SocketIO-based web frontend. The load balancer keeps track of the storage and execution load of each node and assigns new project storage requests and manages replicas if configured.
- *Vision node*. A Vision node contains a list of projects, stored in the form of folders similar to a Git[5] repository (see Figure 2). Each project contains a list of user-uploaded artifacts, extracted information, and analysis reports. Associated with a given project folder is a Python-based API (Application Programming Interface) for asset uploading, unpacking, disassembling, and running analyses. This API is served through a remote proxy and load balancer. Since a project is physically stored in a folder, its size must fit into a single node. A Vision node also contains a list of Huey workers[6]. These executors will receive asynchronized job requests and directly operate on the projects stored in the same node. Job requests will be assigned to the executors that are on the same node of the target project. However, if all the executors on the given project's node are busy, a job request will be assigned to a worker on a different node and the worker will access the project by calling remote procedures through the proxy.
- *GPU-enabled Vision node*. Vision nodes under this category have one or more GPU devices available for machine learning-based analytic tasks. However, GPU-related tasks will have the priority to take over the GPU-enabled job workers.

The second consideration is about user interface adaptability and responsiveness. In our past solutions, we generated a single report in the format of a JSON document and rendered it directly through a website. This is also the typical practice used in existing CTI online platforms for malware analysis and triage, such as VirusTotal[7] by Google and Hybrid Analysis[8] by CrowdStrike. As internet connections speed have been significantly improved, the attackers start to increase the size of their malware artifacts. We have seen certain samples exceeding 200 MB in size. The analysis report for these samples can exceed 1 GB. Rendering a 1 GB JSON document with diagrams and visualizations on the client-side is infeasible. The existing frameworks simply skip files of a certain size. For example, Hybrid Analysis has a maximum file size of 100 MB. To address this issue, we represent each analysis report as a SQLite[9] database-backed by a single file. This way, we can load and render specific information on the web interface as needed. We use SocketIO[10], a room-based event-driven framework, as the channel for client-frontend and client-Vision communication to transfer information about the project and the analytic report.

The third consideration is plug-in integration. Direct integration of reverse engineering tools such as IDA Pro[11] and Ghidra[12] is highly convenient for malware analysts, as they can obtain the analytic report on the fly while they are investigating certain malware campaigns. However, these disassemblers come with

---

[5] Git source code version control framework

[6] Huey: a lightweight job executor framework.

[7] VirusTotal: a free virus, malware, and URL online scanning service.

[8] Hybrid Analysis: a free malware analysis service.

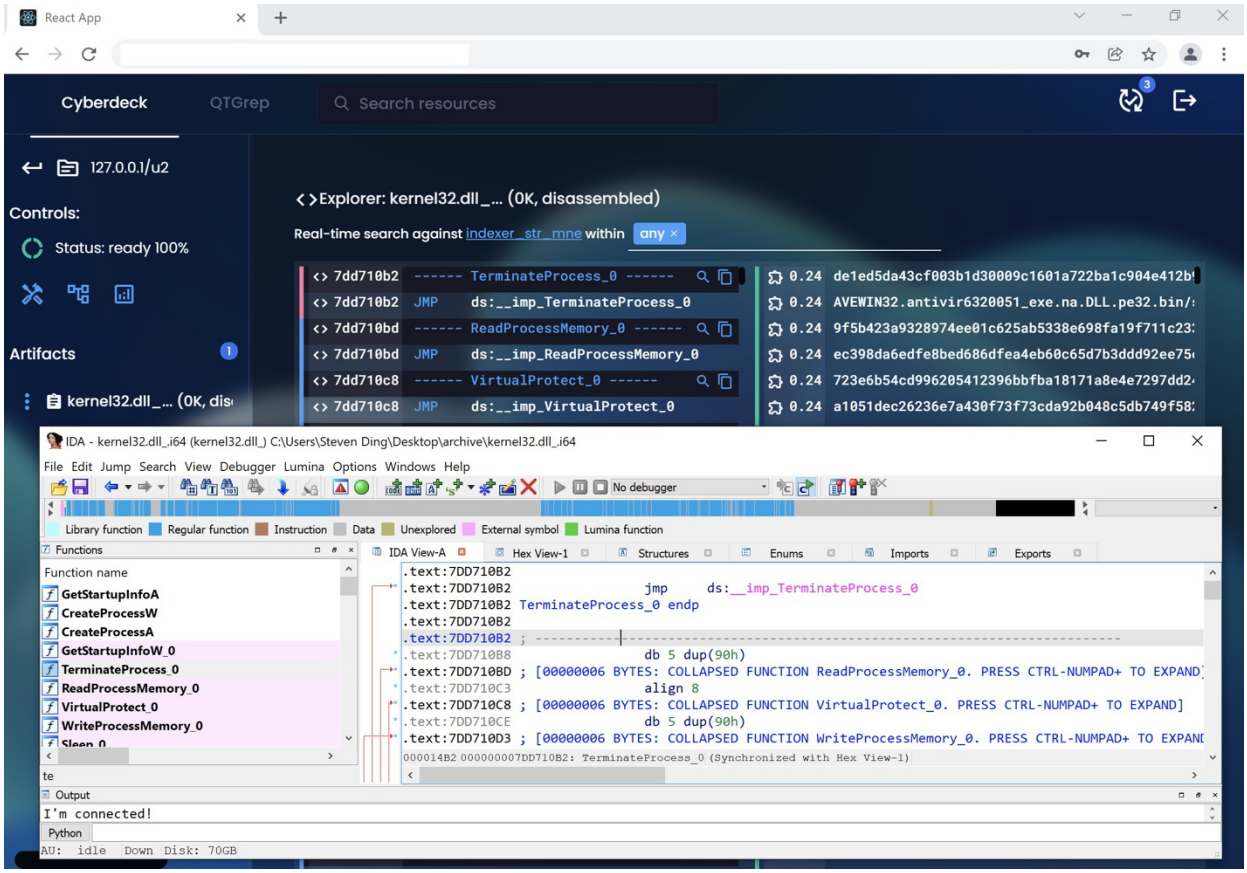
[9] SQLite Database: a small, fast, and self-contained database.

[10] Socket.IO: Bidirectional and low-latency communication.

[11] IDA Pro: an interactive disassembler.

[12] Ghidra: A software reverse engineering (SRE) suite of tools.

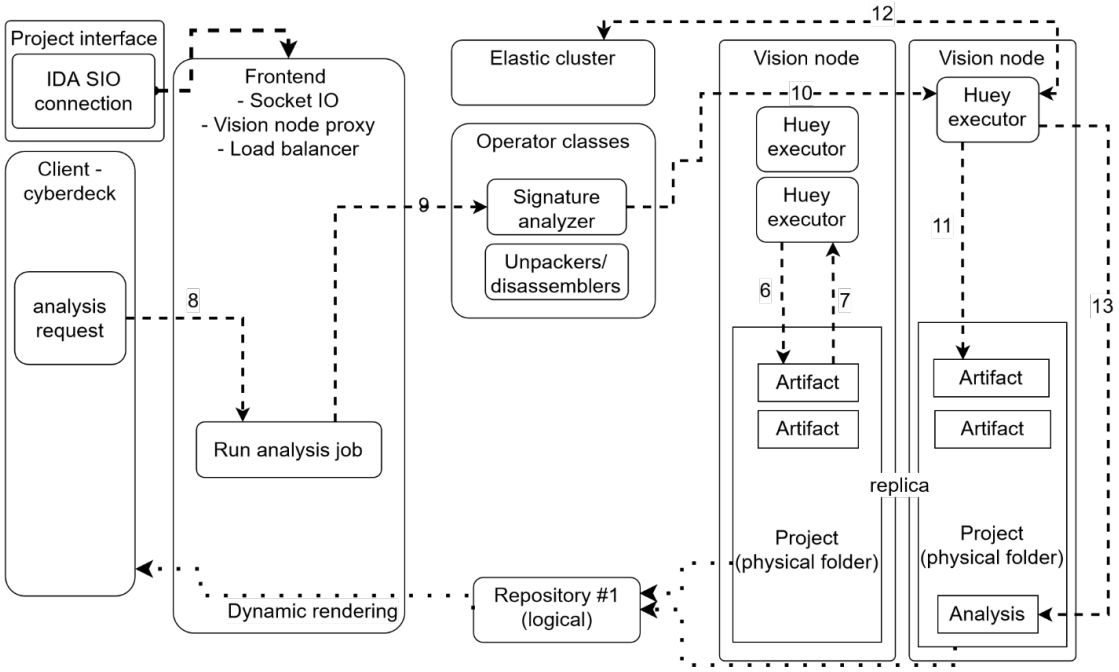
Figure 3: IDA Pro plug-in connected to the frontend service and shown up on the web interface. The white window at the front is IDA Pro and the browser window at the back is the JARV1S web UI. Both show the same information synced by SocketIO



different plug-in requirements, especially regarding the integration of user element components. These requirements and related APIs also frequently change. Instead, we propose to connect the disassembler plug-in to the web frontend as a SocketIO-based remote procedure call following the same project API interface we used in the Vision node. The user will directly use the web UI (User Interface) of the frontend service to conduct analysis and access results (see Figure 3). Therefore, the project-based analytic and data retrieval interface is backed by three different implementations:

- *Python-based implementation that directly accesses the stored data in a folder*. This implementation is used for the data-locality mode where the job worker is assigned to the same Vision node than the stored project folder.
- *RPC-based object proxy implementation*. The executed code is still the same as in the previous implementation, except that the APIs are called from a different Vision node than the one that stores the project folder. It is triggered when the job worker is in a different node than the project storage folder.
- *SocketIO-based RPC object proxy implementation*. This is used as the plug-in-frontend and plug-in-Vision communication channel. It is separated from the previous one, since the communication channel to the client and plug-in is untrusted and needs authentication and certain protections. This way, the frontend can directly call a project API to retrieve information for web rendering. For

Figure 4: JARV1S system flow diagram for the malware decomposition process



example, request the list of function names of a certain range and render them into the user's browser. The Vision node can also connect to the same SocketIO room, identified by the user's unique id, and access any of the SocketIO-connected plug-in clients.

3. Malware Decomposition

The clone search-based malware decomposition method consists of two procedures: indexing and analyzing. Figure 2 shows the steps involved for a user to index an artifact or a project which contains a list of artifacts, specifically:

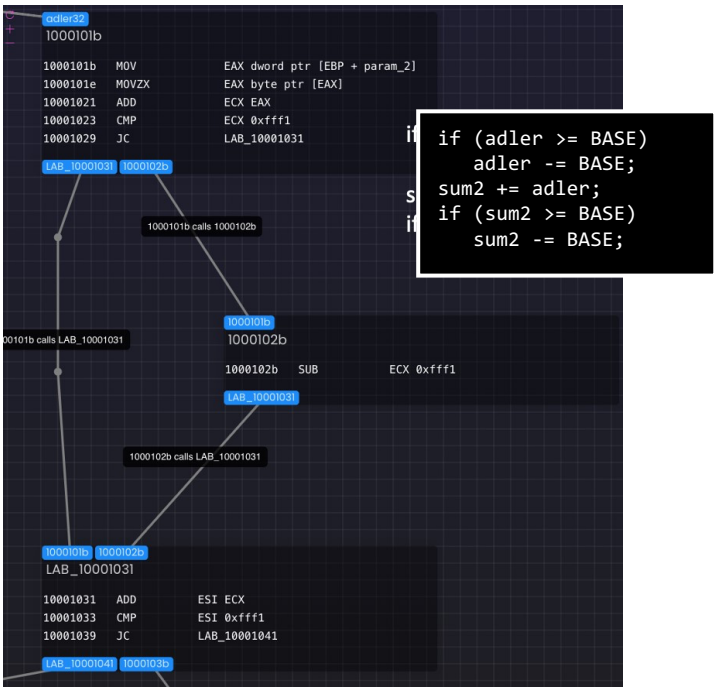
1. The user submits one or more malware samples through the web browser into an existing or newly created project. The new file event is triggered within the SocketIO room specifically for that user. The samples are transferred to the frontend SocketIO server in small chunks.
2. Upon receiving the complete file samples, the frontend server resolves to the Vision cluster load balancer and RPC proxy. Files are then transferred to the actual assigned Vision node, compressed, and stored in the corresponding project folder.
3. The user submits an extraction request through the client interface. The SocketIO server triggers the event for extraction execution.
4. The triggered event is mapped into the operator classes for all the applicable extraction steps needed for the target resources. These operator classes include extraction, unpacking, disassembling, and optionally decompiling.
5. The mapped operator classes and the specific job configuration will then be submitted into the Vision cluster, through a load balancer that considers both data locality and individual workloads.

6. The assigned job executor starts the communication with the project folder through the local project API, a SocketIO-backed project API, or an RPC-backed project API.
7. The job executor reads the necessary information about the artifacts and writes back the extracted information to the project folder. The disassembly code, strings, constants, and other common attributes about a malware sample are written into a SQLite-backed single-file database, with a customizer compression dictionary. It should be noted that each executor will be assigned with a single artifact for extraction, rather than all the artifacts to increase the degree of parallelism.
8. Once the extraction process is completed, the user can submit an indexing job through the web user interface on the client.
9. Like the previous user actions, a new event specifically for the indexing job is fired from the SocketIO interface. The event is then mapped into the corresponding analyzers under the operator classes. These analyzers define additional extraction and analysis steps needed before making an artifact searchable.
10. The analyzer class and the configurations will be submitted to the Vision node. A job executor will be allocated for the submitted task. It should be noted that to provide efficient access and search, certain frequently accessed projects will be replicated across different nodes through synchronized project API transactions, to increase data parallelism. Replications are designed to be automatically provisioned. Once a project is accessed less frequently, its replica will be removed.
11. During the decomposition process, the analyzer will extract observable characteristics about code, data, and strings from the disassembly code. It will then submit the extracted phenotypes as basic indexable elements into a separate Elasticsearch cluster.
12. Finally, the job executor updates the project status and indicates that it is completed.

To conduct a decomposition analysis against a specific project, a similar list of steps is followed, as shown in Figure 4. The user first submits an analysis request through the user interface (Step 8) targeting a specific project. For new artifacts that do not yet exist in the system, the user follows Steps 1-7 from Figure 2 to create a new project and add new artifacts. It should be noted that the target to be indexed does not have to be indexed. After Step 8 (Figure 4), the job analysis event is fired, mapped into the corresponding analyzer classes, and submitted to the Vision cluster requesting a job executor (Steps 9-10). Once assigned, the executor will extract phenotypes and start the decomposition algorithm. The analytic result is written into a single SQLite-backed database file (Steps 11-13).

The proposed functional decomposition method is a three-step process: 1) given a target malware sample, we identify the list of assembly functions. 2) For each assembly function, we search for its clones against a collection of benign software binaries, to remove some commonly reused library code. 3) For the assembly functions that do not match any existing benign library functions, we conduct another round of clone search against the malware in the repository. Overall, it combines and repeats basic assembly code clone search at the function level. To search for clones of given assembly functions, we opt for a signature-based approach. As we mentioned earlier, we extract phenotypes, observable characteristics from the assembly code, to match functional level clones.

Figure 5: Visualizing the control flow graph of a function in JARV1S. This example shows that 0xfff1 is a critical numeric literal for the adler32 function. The white text is the original source code



- Assembly code operation n-grams. Given a piece of assembly code, we extract the list of assembly operations, such as *mov*, *add*, *push*, and *pop*. Then, we extract sequential *n* operations as an *n*-gram single phenotype. For example, *(mov, add, add)* is a 3-gram phenotype. We use both 2-grams and 3-grams. We have chosen these sizes for n-grams based on results by Khoo *et al*. [14] and our empirical evaluation.

- Referred string constants and stack strings. Given a piece of assembly code, we identify a list of referred string constants as phenotypes. We investigate the data reference operations and operands in the assembly code, such as data loading from a specific segment. Given the located data reference address, we scan and verify if there is a valid UTF-8 or ASCII string present by checking if the encoding is valid. For malware samples, static strings are often obfuscated as a stack string, where each character of the string is stored separately and combined at run time. We identify short strings from a single basic block and merge them into a single string value.

- Numeric literals. Numeric literals are useful to match different implementations or variants of the same compression or encryption algorithm. Figure 5 shows an example of the adler32 function, where the 0xfff1 constant is critical for its control flow. We extract all the referred constant numbers from the given piece of assembly code. Numeric literals are of different forms in assembly code. The same number can be encoded in a different format or be of distinct size. We normalize the numbers according to the constant type. For normal integer numbers with a base of either 2, 8, 10, or 16, we convert them into integer values of base 16 in the form of a hexadecimal string. Additionally, we also include its original form in case it is part of the disassembler analysis comments, where proper reference resolution and value normalization have already been completed. For floating-point values, it is difficult to match the same numbers that are encoded with different precision schema. Instead,

we convert each floating-point number into two hex strings, one for single-precision encoding and one for double-precision encoding, as the phenotypes match to any format.

Given a target assembly function, we extract a set of phenotypes and match it against the repository by combing the Okapi bm25 function and the TF-IDF function [10]. The Okapi bm25 scoring function is popular for text data retrieval, where each piece of text data to be retrieved is represented as a set of keywords. Given a query Q and a candidate document D, the similarity score can be formulated as:

$$\mathrm{bm25}(D,Q) = \sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b\cdot|D|/\mathrm{avgdl})} \tag{1}$$

This function is a cumulative sum of independent scores evaluated for each of the n query terms q_i. In our context, it is one of the extracted phenotypes, such as a single operation 2-gram. For each phenotype q_i, IDF(q_i) measures the invert of the total number of assembly functions that have q_i. A common phenotype q_i which is presented in many assembly functions will thus contribute less to the total score. $f(q_i, D)$ measures the frequency of phenotype q_i in the candidate function D. The more frequent the phenotype is present in the candidate, the more representative it is for that candidate. The frequency is normalized by the relative length of the given candidate function compared to the average length of the assembly functions in the repository. k_1 and b are two hyperparameters controlling the degree of normalization. *avgdl* is the averaged document length in the database. A longer candidate function D tends to have higher frequency counts for the extracted phenotype. This is especially important to match for certain compiler optimizations, where loops are unrolled. k_1 and b control the degree of normalization strength at the query time.

Given a query Q, the scores for all the candidates D are relative scores. It means that the candidate's score can only be compared within the result for the same query and its range is unbounded. To bound the score into a human-friendly similarity score of range 0 to 1, we further normalize the score and divide a candidate's score by a bm25 score of matching the query itself.

$$\mathrm{bm25n}(D,Q) = \frac{\mathrm{bm25}(D,Q)}{\mathrm{bm25}(Q,Q)} \tag{2}$$

This way, the candidate score is bound to 1 and it is comparable across different queries. bm25 is the default scoring function for Elasticsearch and it is ideal for situations where the queries are mostly shorter than the indexed documents. This is quite a common situation for text data retrieval in search engines. The queries mostly consist of only a few words and in contrast, the retrieved documents are much longer, such as a news article. In our use case, for phenotype matching on assembly clone search, bm25 is a useful scoring function against compiler optimization techniques such as loop unrolling and inline function calls, where the query is indeed much shorter than the correct candidate to be matched.

However, there are other situations where the bm25 function does not fit well. If the query and the candidates are of comparable length, bm25 may not be ideal, due to the normalization term for D. For text data retrieval, the TF-IDF scoring function is more applicable for this scenario. It can be formulated as:

$$\mathrm{tfidf}(D,Q) = \sum_{i=1}^{n} sqrt(f(q_i,D)) \cdot log(\frac{DF+1}{\mathrm{DF}(q_i)+1}) \cdot \frac{1}{sqrt(|D|)} \tag{3}$$

Where q_i D, and $f(q_i, D)$ are the same as above. DF and $DF(q_i)$ are the document frequency and frequency of sample q_i in the document respectively. Like the bm25 score function, the TF-IDF score function is also an unbounded relative score. Following the same approach, we normalize the TF-IDF score by dividing it with a self-match TF-IDF score:

$$\text{tfidfn}(D, Q) = \frac{\text{tfidf}(D, Q)}{\text{tfidf}(Q, Q)} \tag{4}$$

Finally, we combine both scoring functions into a unified similarity score for candidate ranking by considering their discriminative power in the retrieved candidates:

$$d(Q, s) = D_{KL}(\max_k\{s(d_i : i = 1..|\mathbb{D}|)\} \;||\; \underbrace{\{1, 0, 0, ..., 0\}}_{k \text{ elements}}) \tag{5}$$

$$D_{KL}(a \;||\; b) = \sum_i a_i \cdot log2(a_i/b_i) \tag{6}$$

Given a query Q, we measure the discriminative power of $s \in$ {bm25n, tfidfn} by comparing the KL-divergence between the top-k retrieved candidate scores, where k is the user-defined parameter controlling the maximum number of the returned results, and a one-hot vector of k elements where only the first element is 1. An ideal scoring function in the ideal situation should be able to provide good discriminative power over the list of candidates, while only keeping the first candidate a perfect matching score. Then, we only select the scoring function that yields the lowest value of d:

$$\text{similarity}(D, Q) = \begin{cases} \text{bm25n}(D, Q), & \text{if } d(Q, \text{tfidfn}) > d(Q, \text{bm25n}) \\ \text{tfidfn}(D, Q), & \text{otherwise} \end{cases} \tag{7}$$

As discussed earlier, the decomposition process consists of three main steps. Algorithm 1 provides the details of each individual step. Line 1 corresponds to Step 1 for assembly function extraction and retrieval.

Algorithm 1 Functional decomposition of a sample s by clone search

1: $\mathbb{Q} \leftarrow \text{extract}(s)$ ▷ Retrieve assembly functions from the artifact s after disassembling
2: unknown = {}
3: **for** $Q \in \mathbb{Q}$ **do** ▷ Loop through each extracted function Q.
4: $m \leftarrow max_1\{\text{similarity}(Q, D) : D \in \mathbb{B}\}$ ▷ Clone search against benign set.
5: **if** m != 1 **then** ▷ If m equals 1, we found a known benign function.
6: unknown = $\{m, ...\text{unknown}\}$ ▷ Store the unknown function.
7: **end if**
8: **end for**
9: decomposition $\leftarrow$ {} ▷ The decomposition result mapping.
10: **for** $u \in$ unknown **do** ▷ Loop through each unknown function u.
11: $m \leftarrow max_k\{\text{similarity}(u, D) : D \in \mathbb{M}\}$ ▷ Clone search against the malicious set.
12: **for** $r \in m$ **do** ▷ Loop through each candidate.
13: **if** $r.score == m_0.score$ **then** ▷ Consider entries having the same top score.
14: $p \leftarrow \text{project}(r)$ ▷ Retrieve candidate's project ID.
15: decomposition$[p] = (u, r)$ ▷ Store the entry result entry under given project.
16: **end if**
17: **end for**
18: **end for**
19: return decomposition

Lines 2-8 correspond to Step 2 for clone search against a list of benign artifact projects. The goal is to first remove any known functions commonly existing in the benign executables. Lines 10-18 correspond to Step 3, where we search for the unknown functions against the malware projects.

4. EXPERIMENTS

Figure 6: Similarity matrix regarding the match ratio in the zero-day family testing set

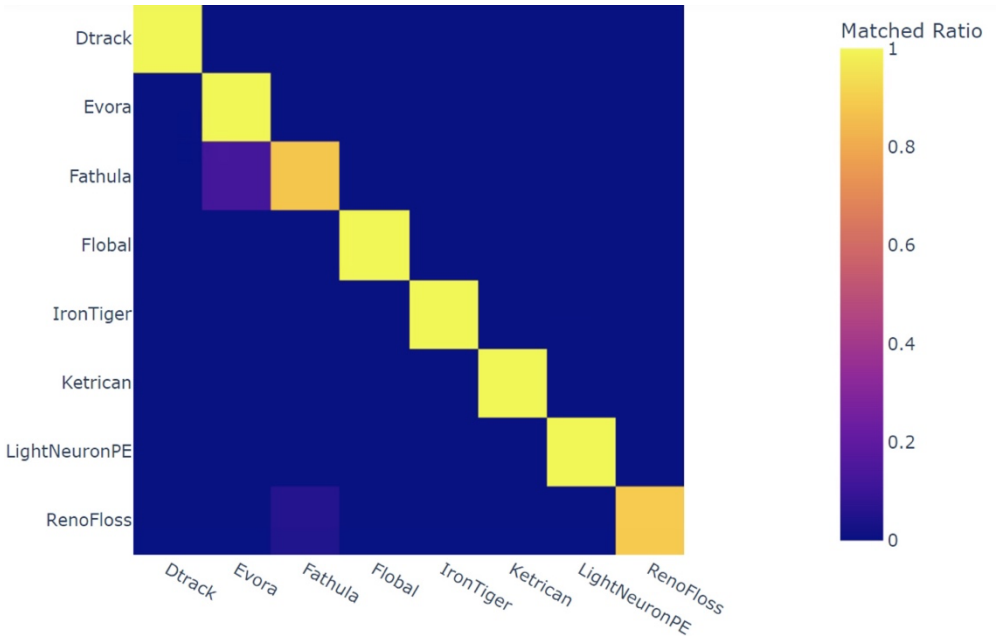
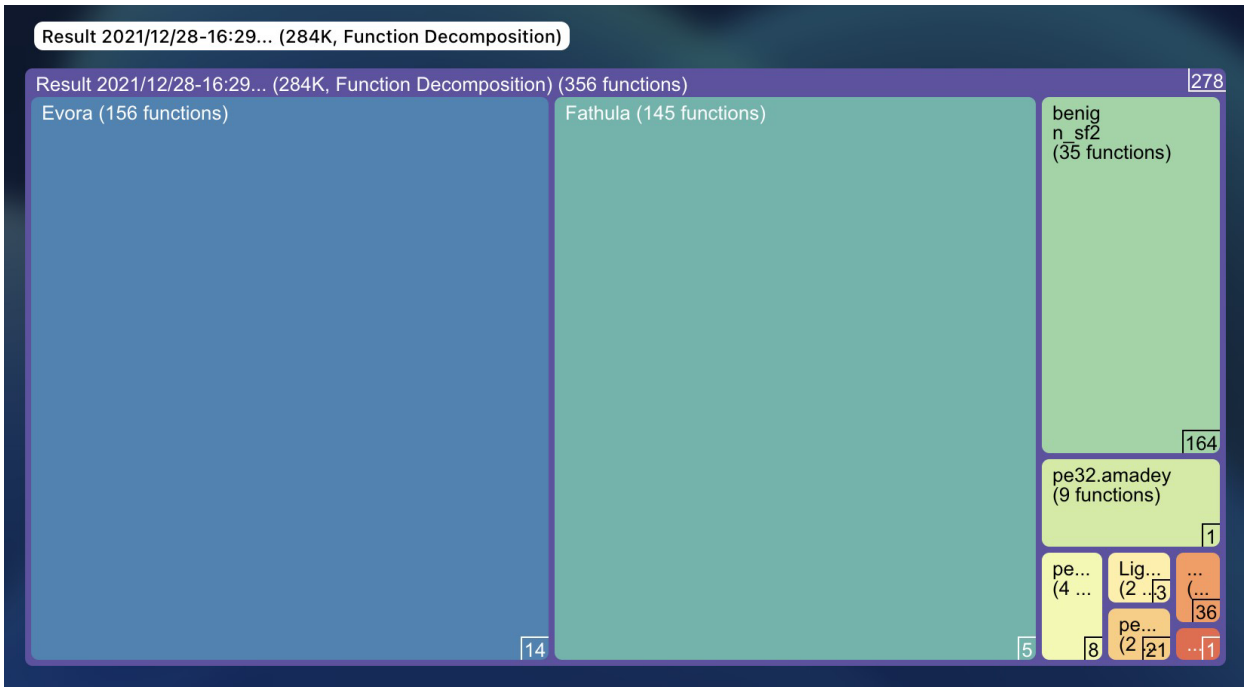


Table I: Some sample malware families and benign software categories used in the experiment

Dataset	Malware Families
Malware Repository	Agent Tesla, Formbook, GuLoader, NanoCore, RemcosRAT, MASS Logger, NjRAT, Dridex, Quakbot, Gozi, AveMariaRAT, Snake Keylogger, IcedID, AsyncRAT, RedLine Stealer, HawkEye Keylogger, Cobalt Strike, RaccoonStealer, NetWire, ModiLoader, ...
Benign Repository	Winamp, Mozilla Firefox, Maxthon, SeaMonkey, Windows Live Messager, Pidgin, ESET Nod32, Defraggler, Avant Browser, TortoiseSVN, Picasa2, Google Talk, VirtualBox, Thunderbird, Silverlight, PowerDVD, FastStone Image Viewer, PowerISO, Flash Player, 7z, MySQL, Blender, XBMC Media Center, Win7codecs, Tera Term Pro, ...
Zero-day Set	Dtrack, Evora, Fathula, Flobal, IronTiger, Ketrican, LightNeuronPE, RenoFloss

To test our design, we created a repository of 200,000 malware samples with 100,000 benign samples. Malware samples are grouped based on their family and the samples of the same family are stored in the same Vision project. All benign samples are stored in the same project. It takes around 43 hours in total for unpacking, disassembling, and indexing all the samples. After, we use a separate set of malware

Figure 7: Tree map visualization of the analytic summary for a sample from the Evora family

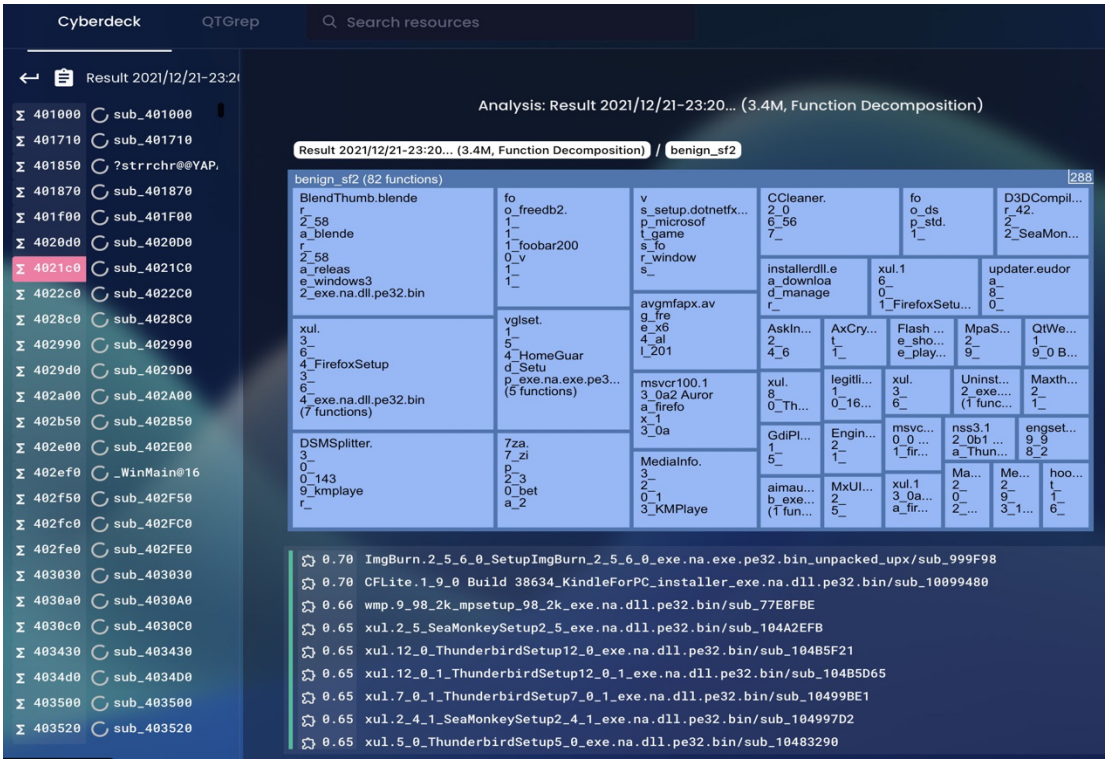


families to simulate zero-day malware families and samples that were not indexed into the system. The malware repository contains 394 families and the repository of benign binaries contains 4,098 categories. Table I lists some of them. To simulate the zero-day malware families, we use a list of families that are not indexed in the repository.

For each sample in the zero-day malware family set, we search it against all the samples including the benign repository, the malware repository, and the zero-day set using the malware decomposition analysis. Given an analytic result, we remove all the detected benign assembly functions and check what is the majority family the rest of the assembly function belongs to. Figure 7 shows an example. There are 356 functions in total, after removing the 35 benign functions. 156 functions matched to the Evora family and 145 functions matched to the Fathula family (no overlapping functions). Therefore, the top matched family for this sample is Evora. Given all samples of a malware family, we estimate the ratio of samples matched for each of the zero-day families. Figure 6 shows the matching ratio across different families and across the zero-day set. It shows that the majority of the malware families were all correctly matched (100% ratio). There are two false negatives (15%) for the Fathula family and one false negative (5%) for the RenoFloss family.

Besides the tree map view shown in Figures 1 and 7, our UI interface also allows the user to further explore the detected cloned functions (Figure 8). The user can browse the list of assembly functions (left panel). After clicking on one of the entries, the user can see the list of detected function clones on the lower right panel. The user can also compare and see the differences between the selected assembly function with a candidate function by clicking on one of the entries. The differences are visualized either using a typical text comparison method or combined with a graph-based comparison method. Additionally, the user can click on any of the rectangles shown on the tree map to further break down the detected clones of a specific

Figure 8: The user interface for rendering our decomposition analysis result



family to see how the clones are distributed across samples in a specific family. The upper-right panel in Figure 8 shows an example of clicking into the benign group and seeing how the functions of different benign samples can be matched.

5. Related Works

Malware similarity analysis is the area of researching methods for matching malware variants. Similarity analysis research has shown multiple different techniques for successful variant matching, such as feature hashing, neural network-based similarity analysis, and clone detection. One method proposed by Jang *et al*. used hashed feature vectors and co-clustering techniques for malware detection and family classification [11]. In a real-world environment, there is no way to predict which features will be significant in the future. This is why JARV1S bases its analysis on assembly code. Neural networks have also been applied to malware family categorisation. Works proposed by Hsiao *et al*., Zhu *et al*., Stokes *et al*. and Khandhar have all used Siamese-based neural network architectures for malware family classification. [8, 7, 6, 9]. Unlike JARV1S, these methods either fail to show any results against zero-day malware or show results with decreasing accuracy. Clone searching is the method of finding duplicate code fragments between software repositories [12]. Clone search for benign software has been researched extensively due to the frequent practice of copying and pasting code with minor change. Recently, clone search has expanded to analysing assembly code [13, 14, 15, 16, 17]. Khoo *et al*. identified code clones using control flow sub-graphs and data constants [14]. Hu *et al*. compared extracted semantic signatures from emulating binary function execution for clone detection [15]. Farhadi *et al*. proposed BinClone, a tool for malware code fragment clone detection from a token-level approach [16]. Ding *et al*. proposed Asm2Vec, a framework to learn the lexical semantic relationships of assembly functions based on

assembly code [13]. Ding *et al*. also proposed the static framework Kam1n0 for assembly code-based clone search [17]. Unlike these systems, JARV1S does not only find the similarity between binaries, but also compares the binaries in behavior and connection to APTs. As well, JARV1S provides a human friendly environment for result analysis. Xue *et al*. and Lee *et al*. have also conducted clone searches on assembly code, but these systems were designed only for finding compromised code within a repository [18, 19].

6. Conclusion

In this paper, we propose a new malware triage system. Given a piece of malware, it can trace the origin of the extracted assembly functions from known binary categories and samples. We propose the concept of phenotype assembly clone search, where one can match different assembly functions based on the observable characteristics. The system is designed with scalability, efficiency, and adaptability considerations. For real world application, JARV1S has been designed for seamless integration into an existing APT processing pipeline. Our example shows that the system is efficient and accurate in analyzing new malware samples in both known and unknown families.

References

[1] A. Mockus, "Large-scale code reuse in open-source software," in *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 7–7.

[2] Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, 2017.

[3] M. Apel, C. Bockermann, and M. Meier, "Measuring similarity of malware behavior," in *Proceedings of the 34th Annual IEEE Conference on Local Computer Networks, LCN 2009, 20-23 October 2009, Zurich, Switzerland, Proceedings*. IEEE Computer Society, 2009.

[4] B. Lau and V. Svajcer, "Measuring virtual machine detection in malware using DSD tracer," *J. Comput. Virol.*, vol. 6, no. 3, 2010.

[5] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu, "Combining file content and file relations for cloud-based malware detection," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, C. Apt´e, J. Ghosh, and P. Smyth, Eds. ACM, 2011.

[6] J. Zhu, J. Jang-Jaccard, and P. A. Watters, "Multi-loss Siamese neural network with batch normalization layer for malware detection," *IEEE Access*, vol. 8, 2020.

[7] J. W. Stokes, C. Seifert, J. Li, and N. Hejazi, "Detection of prevalent malware families with deep learning," in *Proceedings of the 2019 IEEE Military Communications Conference, MILCOM 2019, Norfolk, VA, USA, November 12-14, 2019*. IEEE, 2019.

[8] S. Hsiao, D. Kao, Z. Liu, and R. Tso, "Malware image classification using one-shot learning with Siamese networks," in *Proceedings of Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019, Budapest, Hungary, 4-6 September 2019*, ser. Procedia Computer Science, I. J. Rudas, J. Csirik, C. Toro, J. Botzheim, R. J. Howlett, and L. C. Jain, Eds., vol. 159. Elsevier, 2019.

[9] S. Khandhar, "A few-shot malware classification approach for unknown family recognition using malware feature visualization," 2021.

[10] I. Mogotsi, "Christopher d. manning, prabhakar raghavan, and hinrich schu¨tze: Introduction to information retrieval," 2010.

[11] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011.

[12] H. Zhang and K. Sakurai, "A survey of software clone detection from security perspective," *IEEE Access*, vol. 9, 2021.

[13] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019.

[14] W. M. Khoo, A. Mycroft, and R. J. Anderson, "Rendezvous: a search engine for binary code," in *Proc. of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE Computer Society, 2013.

[15] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, G. Scanniello, D. Lo, and A. Serebrenik, Eds. IEEE Computer Society, 2017.

[16] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *Proceedings of the Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, June 30 - July 2, 2014*. IEEE, 2014.

[17] S. H. H. Ding, B. C. M. Fung, and P. Charland. "Kam1n0: MapReduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 461-470, San Francisco, CA: ACM Press, August 2016.

[18] H. Xue, G. Venkataramani, and T. Lan, "Clone-slicer: Detecting domain specific binary code clones through program slicing," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018, pp. 27–33.

[19] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Proceedings of the 9th International Conference on Internet (ICONI) 2017 symposium*, 2017.