

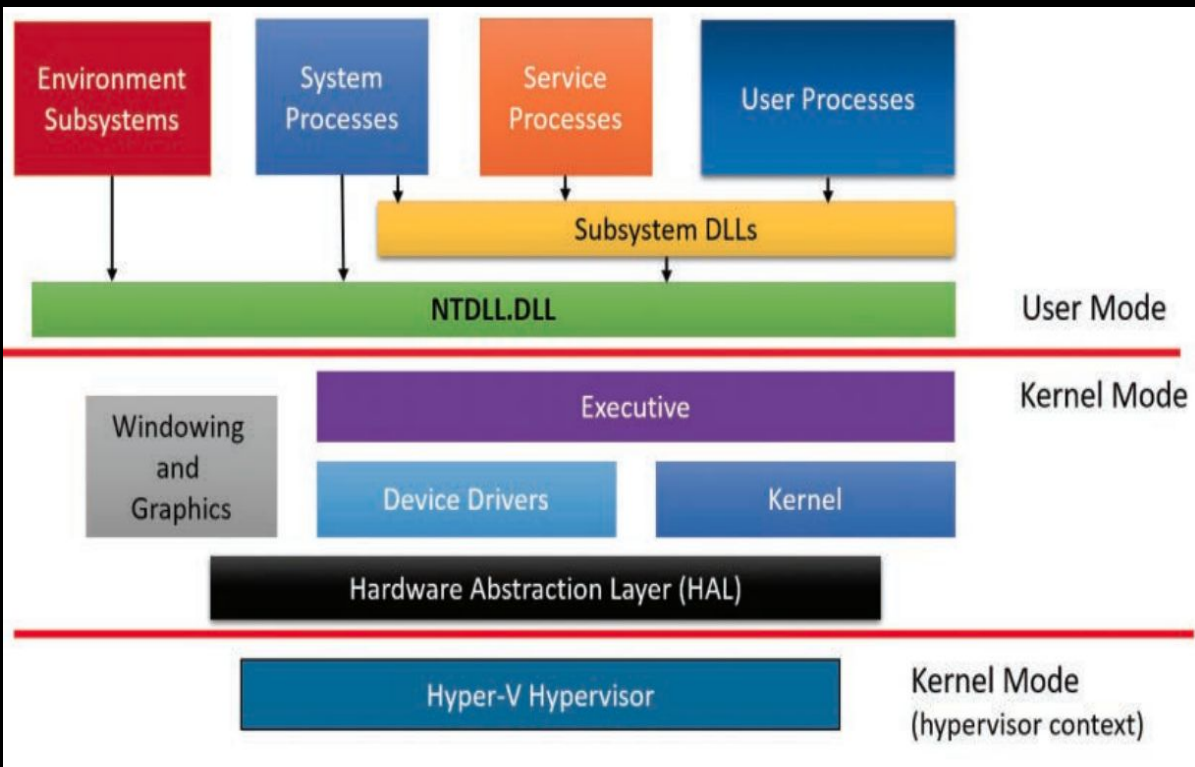
Windows Malware Classification using HNSW and Vector Databases.

Nilesh Jakamputi
Capstone Defense

Agenda

- I. Windows Architecture.
- II. PE File Format.
- III. What is Malware.?
- IV. Obfuscation Techniques.
- V. Dynamic analysis W/ Speakeasy Emulation.
- VI. Preprocessing & Vector Embeddings .
- VII. Methodology.
- VIII. Results.

Windows Architecture



- **Kernel Mode**

- Provides access to privileged instructions such as RDMSR.
- Only trusted code runs here.

- **User Mode**

- Limited Instructions
- Resources provided to processes via system calls to kernel

Windows Architecture

- The Windows O.S segments user space and kernel space based on hardware enabled privilege checks on a per process level.
- A process is a container to separate applications from each other.
 - Each process contains a thread, which is a basic unit of execution.
- A process is created in memory by the windows loader, a thread is created and execution can begin.
- All of which are done via API calls to the kernel.

PE File Format

- The Portable Executable (PE) File format
- A file format for executables, object code, DLLs and others (.sys).
- Contains Headers and Sections
 - Headers details things such as entry points, compile stamps, protections - DOS & NT Header
 - Sections detail things like the byte code, uninitialized data.
- PE file format contains all the information for the OS to load and execute the binary.
- It works on based on RVA- Relative Virtual Addresses or offsets from the base of the file starting with 0x00.

PE File Format

Parse from 0,0 till 3C


At F8, we find the PE header

Disasm General Strings DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports Resources BaseReloc Debug TLS LoadConfig

Offset	Name	Value
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	F8

PE File Format

C++

 Copy

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    WORD                Magic;  
    BYTE                MajorLinkerVersion;  
    BYTE                MinorLinkerVersion;  
    DWORD               SizeOfCode;  
    DWORD               SizeOfInitializedData;  
    DWORD               SizeOfUninitializedData;  
    DWORD               AddressOfEntryPoint;  
    DWORD               BaseOfCode;  
    DWORD               BaseOfData;  
    DWORD               ImageBase;  
    DWORD               SectionAlignment;  
    DWORD               FileAlignment;  
    WORD                MajorOperatingSystemVersion;  
    WORD                MinorOperatingSystemVersion;  
    WORD                MajorImageVersion;  
    WORD                MinorImageVersion;  
    WORD                MajorSubsystemVersion;  
    WORD                MinorSubsystemVersion;
```

Windows Loader

- **Responsibilities of the Windows loader**
 - Reading PE headers, mapping sections, resolving imports, applying relocations
 - Loading and mapping PE files into memory: creating virtual memory mappings for sections
 - Resolving dependencies and importing functions: loading required DLLs and resolving function addresses

What is Malware

- Malware are just programs that have harmful consequences for those that are lucky enough to run them.
 - They come in a variety of forms PE, DLLs, shellcode, powershell, ELF, bash scripts, JS.
- Malwares have different end goals, instead of providing a service to user, they are used for financial gain, espionage or IP theft.
 - They follow the same rules as traditional software - get loaded in memory, call APIs, and execute.
 - They are designed to work on the hardware and operating systems they target.

Obfuscation Techniques

- Operate under O.S rules, but they are designed to circumvent detection measures and confuse security analysts.
 - Xoring the payload with a custom key, AES encrypting the payload using a dynamic seed value.
 - Generating strings to call APIs at runtime.
- Most malware is in userland, kernels usually targeted with a BYOVD, or in rare cases a zero day.

Core Idea: Think of assembly instructions to add two numbers, and think of different ways to achieve that result.

EX:1

MOV ECX, 1

MOV EDX, 1

ADD ECX, EDX

Result -> 2 in ECX

EX:2

XOR ECX, ECX

INC ECX

INC ECX

Result -> 2 in ECX.

EX:3

MOV ECX, 4

DEC ECX

DEC ECX

Result -> 2 in ECX

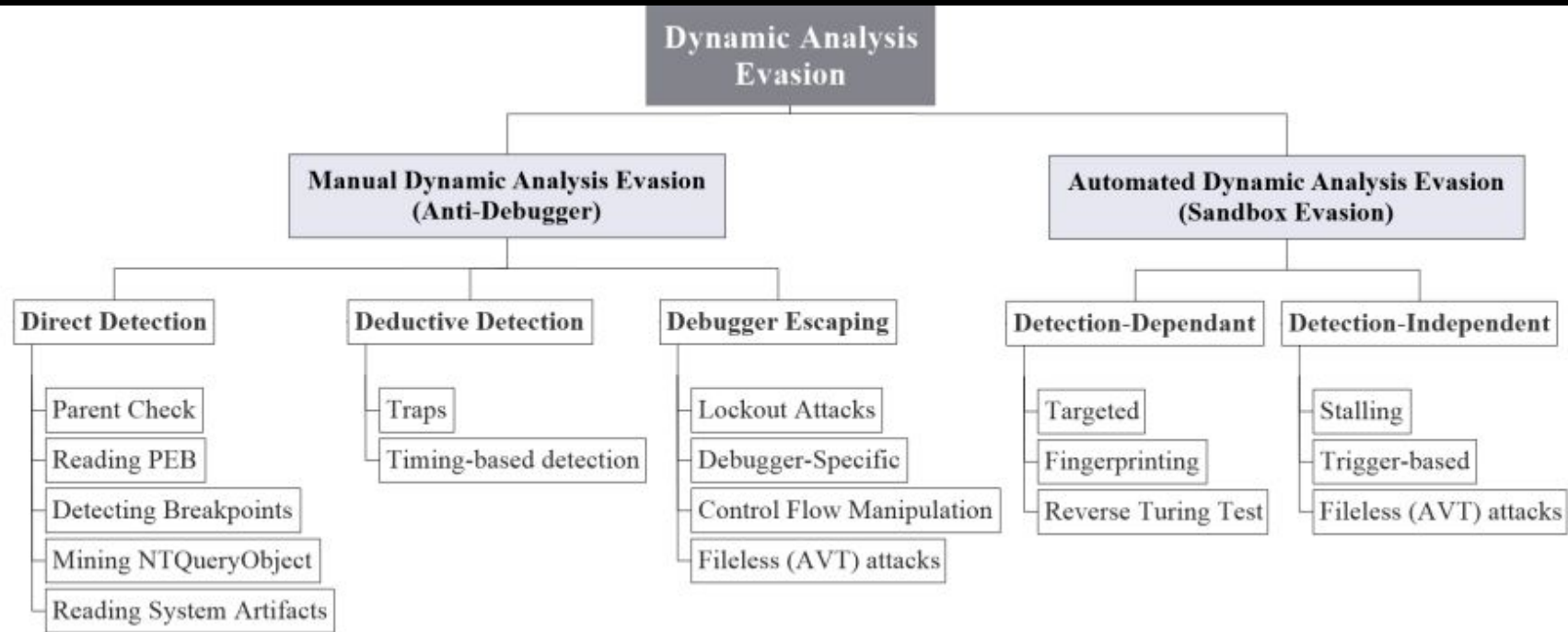
EX:4

MOV ECX, 8

SHR ECX, 2

Result -> 2 in ECX.

Anti Analysis Techniques : Galloro et al. ([source](#))



Dynamic analysis with Speakeasy Emulation.

- An emulator written in python 3 based on the Unicorn Emulation Engine.
- Essentially reads the byte codes and “pretends” to execute
 - Parses the PE file format and all of its headers and sections for an executable.
 - Considers the target instruction set (x86_64), interprets and executes instructions
 - Runs the entire program on unicorn engine without any context, speakeasy provides the context with operating system APIs, objects, running processes/threads, filesystems, and networks.
- The main reason to use this platform is to bypass the aforementioned obfuscation and anti analysis techniques that malware employs.

Dynamic analysis with Speakeasy Emulation.

```
{
  "ep_type": "module_entry",
  "start_addr": "0x40c55a",
  "ep_args": [
    "0x4000",
    "0x4010",
    "0x4020",
    "0x4030"
  ],
  "apihash": "73d36f9847cc19598d07b177f0cec5569d62191ee28210443fea94fb5caf99f5",
  "apis": [
    {
      "pc": "0x41281f",
      "api_name": "KERNEL32.GetSystemTimeAsFileTime",
      "args": [
        "0x1211fd8"
      ],
      "ret_val": null
    },
    {
      "pc": "0x41282b",
      "api_name": "KERNEL32.GetCurrentProcessId",
      "args": [],
      "ret_val": "0xe2b24"
    }
  ]
}
```

JSON Log of an emulated dropper malware.

Each emulation report contains:

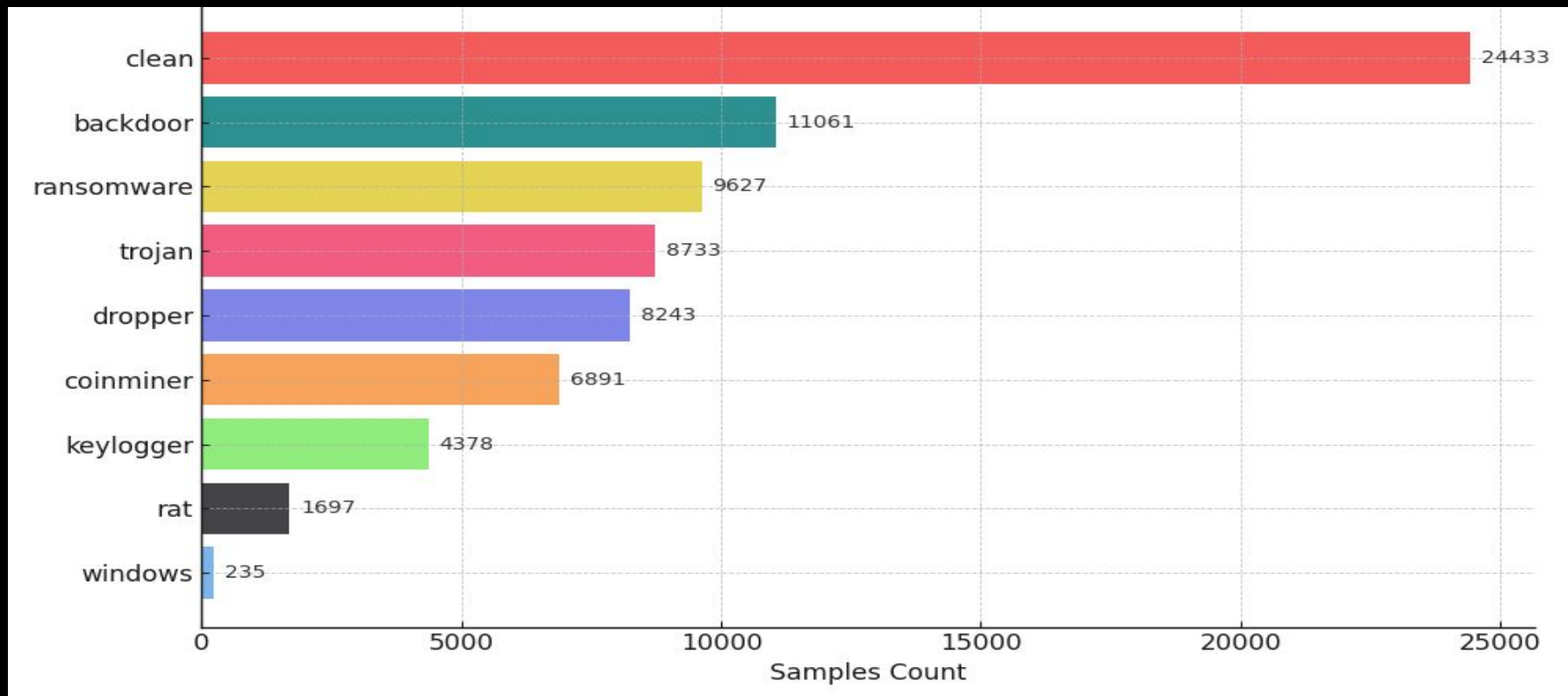
Entry points, Threads, API calls, arguments, and return values.

The speakeasy emulator parses the raw bytes, finds the entry points from the PE headers, allocates spaces and runs the code sections.

Dataset Overview

- Dataset consists of close to 80,000 samples of emulated malware reports.
- Seven different malware types
 - Backdoors, ransomwares, trojans, file droppers, cryptominers, keyloggers and RATs.
- Two categories of clean samples
 - In the wild commonly used software - labeled as clean
 - Covers software such as adobe photoshop, autodesk, blender, and many more
 - Common pre installed windows binaries
 - Notepad, File explorer, Edge etc.

Dataset Overview



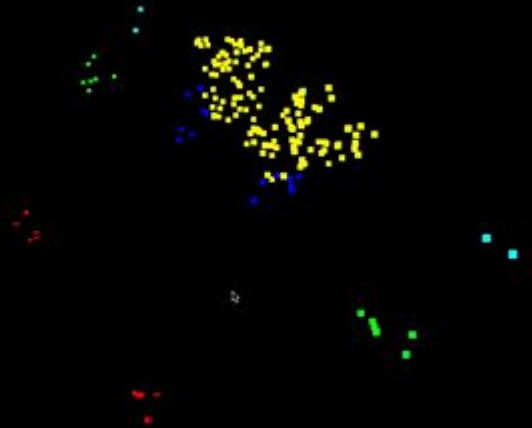
Preprocessing

- The logs from the emulation are preprocessed to reduce vocabulary size and maintain similarity across samples
 - Across malware families, file names, IP addresses, files modified, registry keys accessed vary depending on the task at hand.
 - To remove variance across samples, standardization of the above is required.
 - IP addresses are normalized to <loIP>, <prvIP>, and <pubIP>, based on their classes.
 - Domain strings like thisisnotamalware.ru normalized to (<domain>)
 - File names are normalized to their respective hash functions.
- Preprocessing yields a CSV file, that contains the label and a sequence of API calls, arguments and return values.

Preprocessing

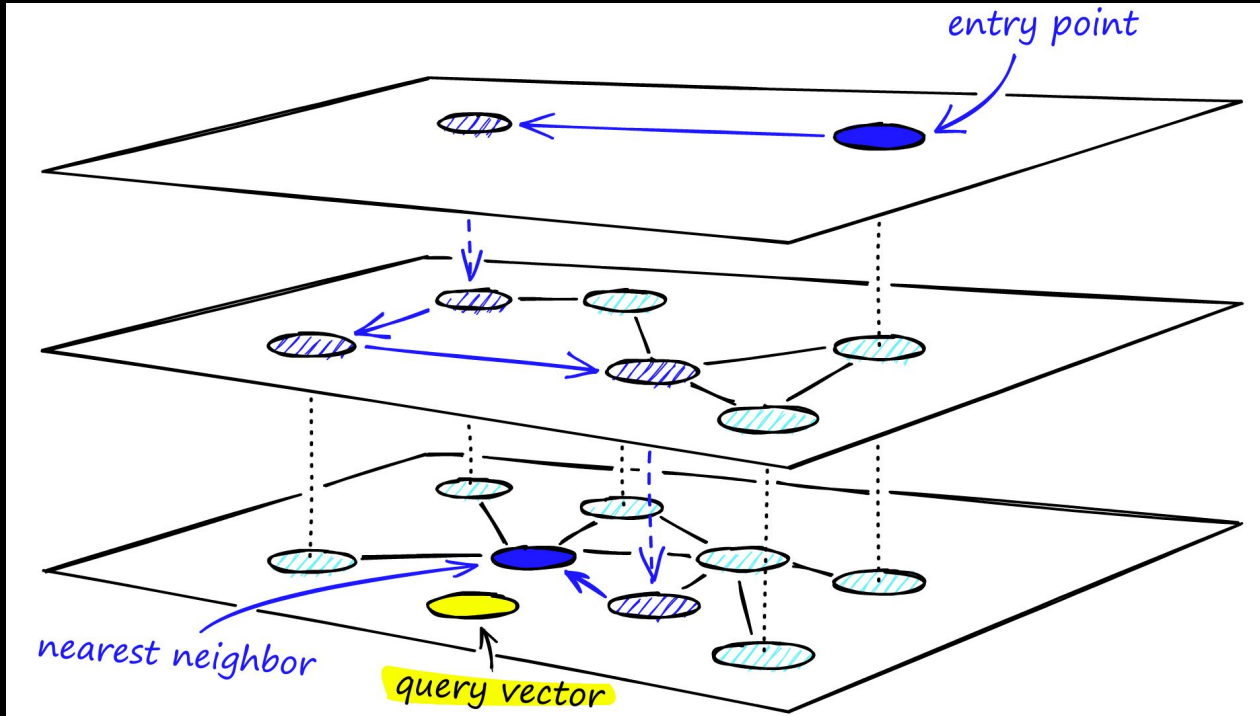
- Our approach chunks and vectorizes each API call, with their corresponding labels and order of appearance using a pre trained sentencepiece transformer model. BAAI/bge-small-en-v1.5 from Hugging face.
- Each word is broken down into tokens, and each tokens vector representations are generated with the transformer model and stored in Qdrant, vector store.

Vector Embeddings



- Each point in the vector space is a unique token, an atomic integer representation of a word.
- Similar words are close to each other. (colour coded)
- It is represented in a 3D space so that we can visualize it
- In reality, each token is represented in 384 dimensions, which impossible to visualize.

Methodology



- We have a vector database for API calls.
- We search it with HNSW.
- HNSW allows us to get an approximate result from a vector query.

Methodology

- The retrieval from Qdrant using HSNW yields the best matched label
 - The retrieval is based on cosine similarity to the closest node found by HNSW
 - Returns one of the seven malware labels or one of two clean labels based on similarity
- We perform reranking of the returned labels using Maximal Marginal Relevance, which diversifies the results and organizes it
- We perform a frequency analysis of the reranked results, and provide a classification

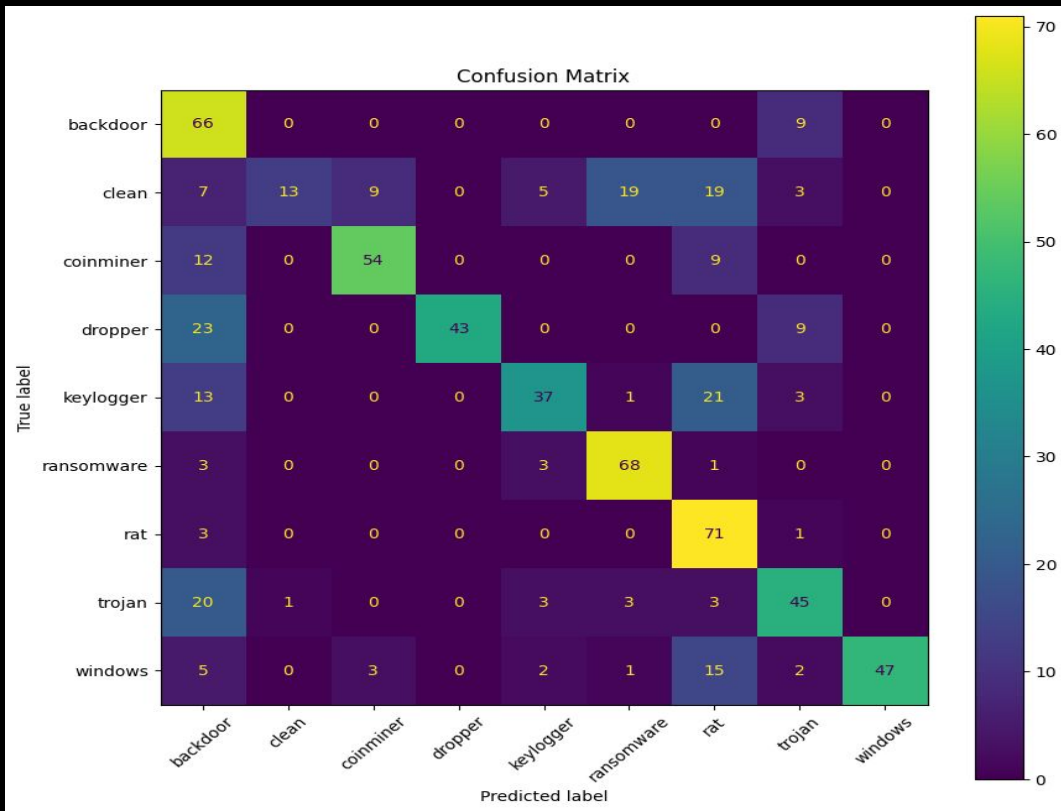
Methodology

- **For Building the database, approximately 8k samples were used.**
 - Maintaining the frequency distribution of the original database
- **For inferencing the database, 225 samples were chosen from the test set.**
 - Each sample was chunked, vectorized and queried.
 - Top 3 results of each API call was taken across the samples and prepared for frequency analysis.
 - Resulting in a collection of 675 documents to analyze. $225 * 3 = 675$

Methodology

- The returned results are analyzed with weights being assigned manually to swing to the malicious way.
 - The weights are heavier for malicious api calls as all of them appear in the clean malware samples as well
 - If not for these weights, everything would be overwhelmingly classified as clean.
 - A clear avenue for improvement.

Results



- Diagonal values indicate successful classification
- Challenges in distinguishing benign and malicious behaviors based on API call frequencies

Results

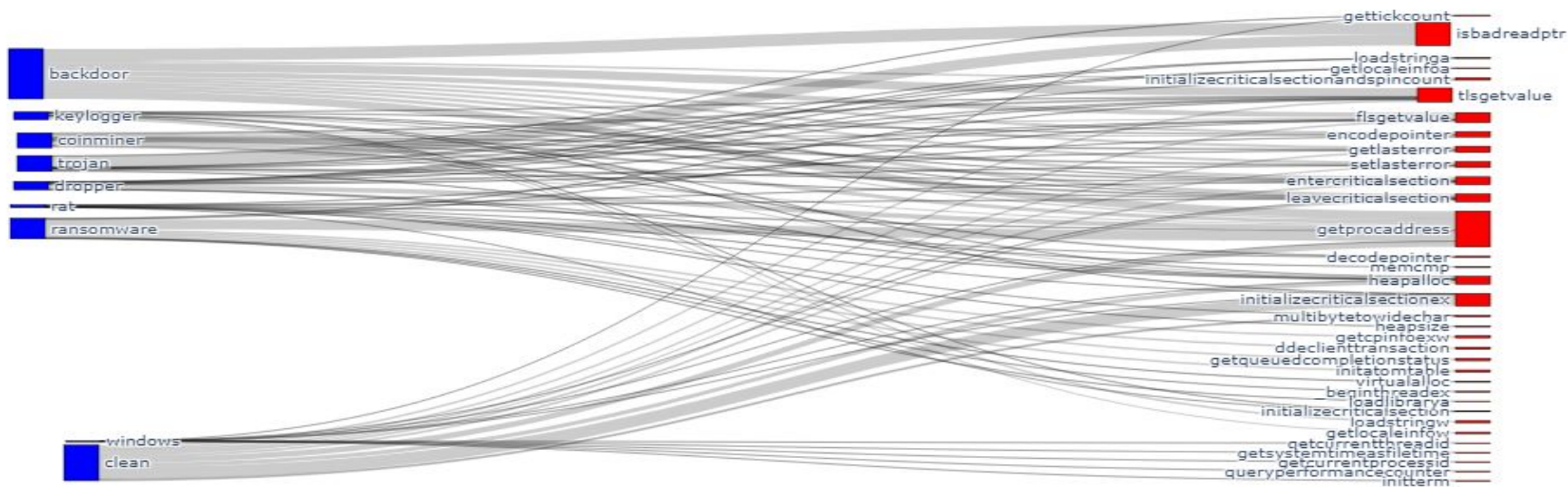
	precision	recall	f1-score	support
backdoor	0.43	0.88	0.58	75
clean	0.93	0.17	0.29	75
coinminer	0.82	0.72	0.77	75
dropper	1.00	0.57	0.73	75
keylogger	0.74	0.49	0.59	75
ransomware	0.74	0.91	0.81	75
rat	0.51	0.95	0.66	75
trojan	0.62	0.60	0.61	75
windows	1.00	0.63	0.77	75
accuracy			0.66	675
macro avg	0.76	0.66	0.65	675
weighted avg	0.76	0.66	0.65	675

Accuracy of 66%
(two-thirds of API
call sequences
classified correctly)

Room for
improvement in
classification
performance

Results

Malware Labels to Top 10 API Calls



Results – Limitations

- High-Dimensional Data Complexity
- Expensive to generate vector embeddings
- Even more expensive to generate context aware embeddings
- Anti Emulation techniques will gain more prevalence
 - Already in the works to evade windows defender and other endpoint detection tools.

Future Work

- **Large Language Models**

- Instead of setting the weight manually, a deep learning framework such as LLMs can be used to set the weights
- This is done by fine tuning a pre trained model to recognize the data we generated from the process above
 - ["input_text": "Classify the following into one of the following classes [clean, backdoor, ransomware, trojan, dropper, coinminer, keylogger, rat, windows], text: windows 1; rat 135; windows 2; clean 4; rat 5; clean 6; coinminer 14; clean 8; clean 30 (truncated for brevity), "output_text": "clean"]
 - We train the model and ask it to classify the sequence for us.
 - But this approach requires a lot data, and more importantly, a lot of compute.

Conclusions

- Malware classification remains a critical challenge in cybersecurity
- Dynamic analysis and vector embeddings offer promising avenues for accurate and efficient malware classification
- We just demonstrate the potential of leveraging API call sequences and similarity search for malware detection
- Continuous research and innovation are essential to combat the ever-evolving landscape of malware threats

Thank you