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

Omid Mirzaei, Roman Vasilenko, Engin Kirda, Long Lu, Amin Kharraz

18th Conference Detection of Intrusions and Malware & Vulnerability Assessment
DIMVA 2021

Motivation

APT Groups Target Firms Working on COVID-19 Vaccines

Microsoft Says Attacks on Seven Companies Blocked

Motivation

Nuclear Weapons Agency Hacked in Widening Cyberattack

Motivation

Google: North Korean hackers have targeted security researchers via social media

Google TAG warns security researchers to be on the lookout when approached by unknown individuals on social media.

Motivation

- Previous efforts to detect code reuse:
  - Binary and code similarity testing
  - Clone detection
  - (Fuzzy) hashing

- Existing approaches are inadequate for these reasons:
  - **Lack of ground truth**
  - **Intense use of evasive techniques**

Outline

- Scrutinizer Overview
- Results
- Discussion
- Conclusion

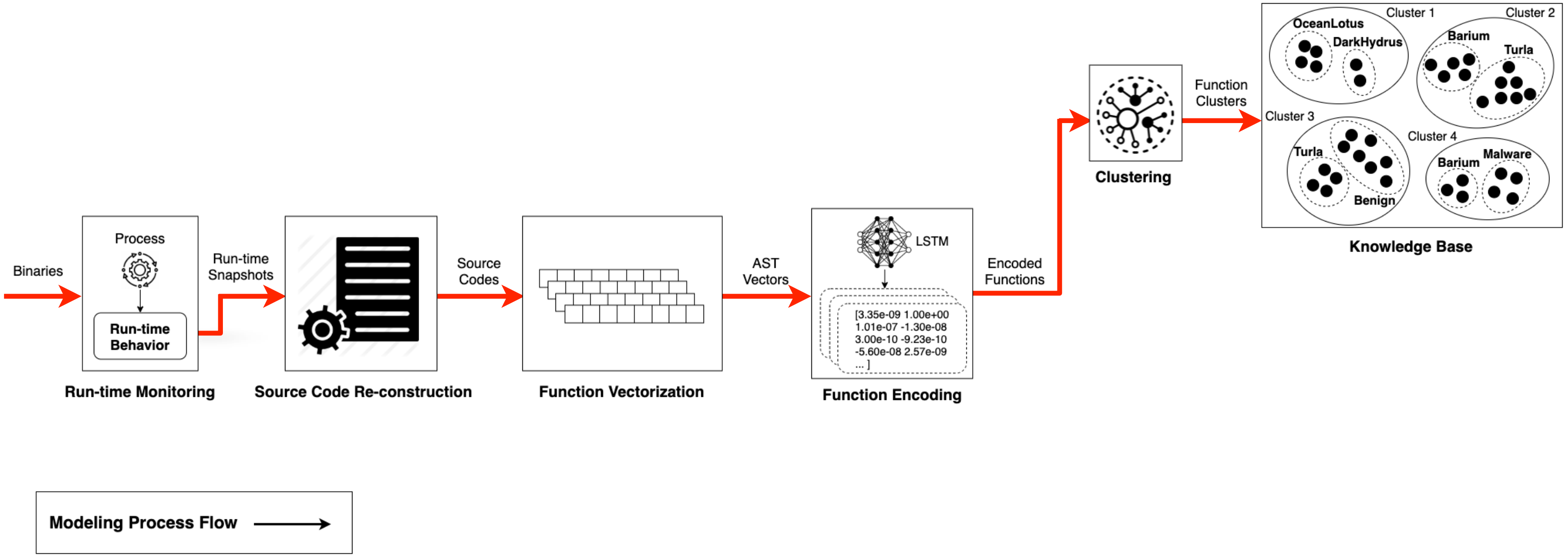
Outline

- **Scrutinizer Overview**
- Results
- Discussion
- Conclusion

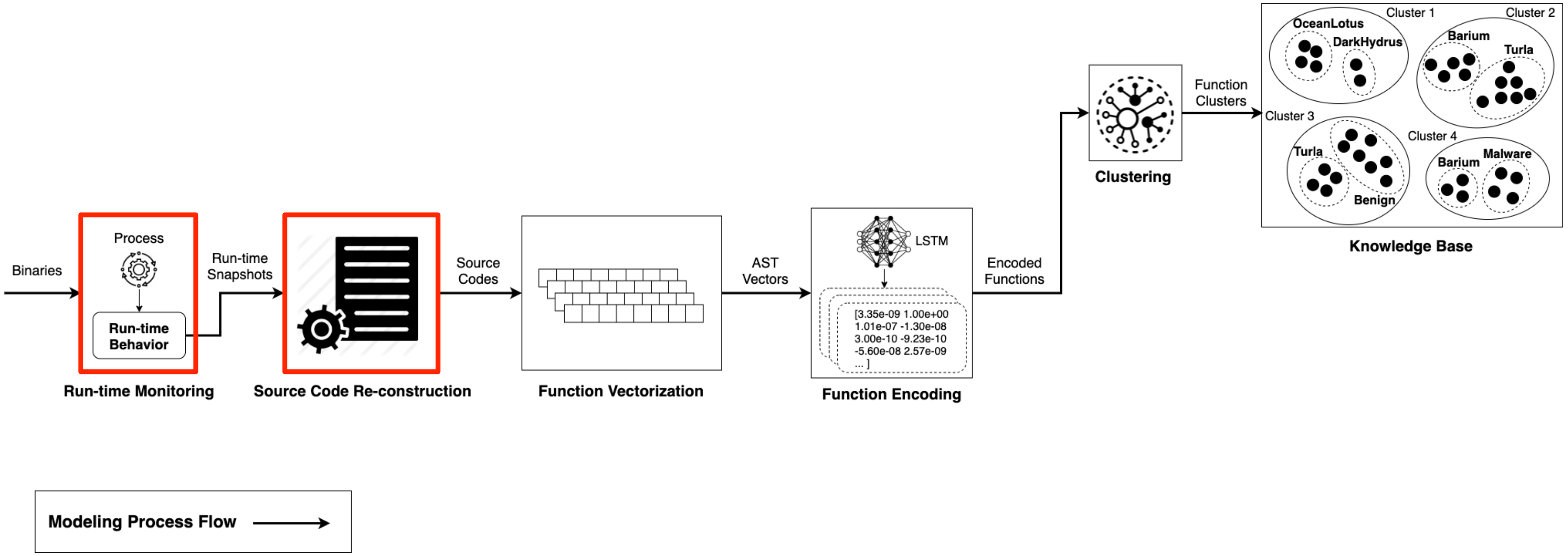
Main Idea

- Identifying code similarities that exist between an unknown sample and those that are known to be used by threat actors from different campaigns
- **Modeling phase**
  - Aim: creating a large knowledge base of previously observed and tagged malware campaigns

Scrutinizer Overview
General Architecture



Scrutinizer Overview
General Architecture



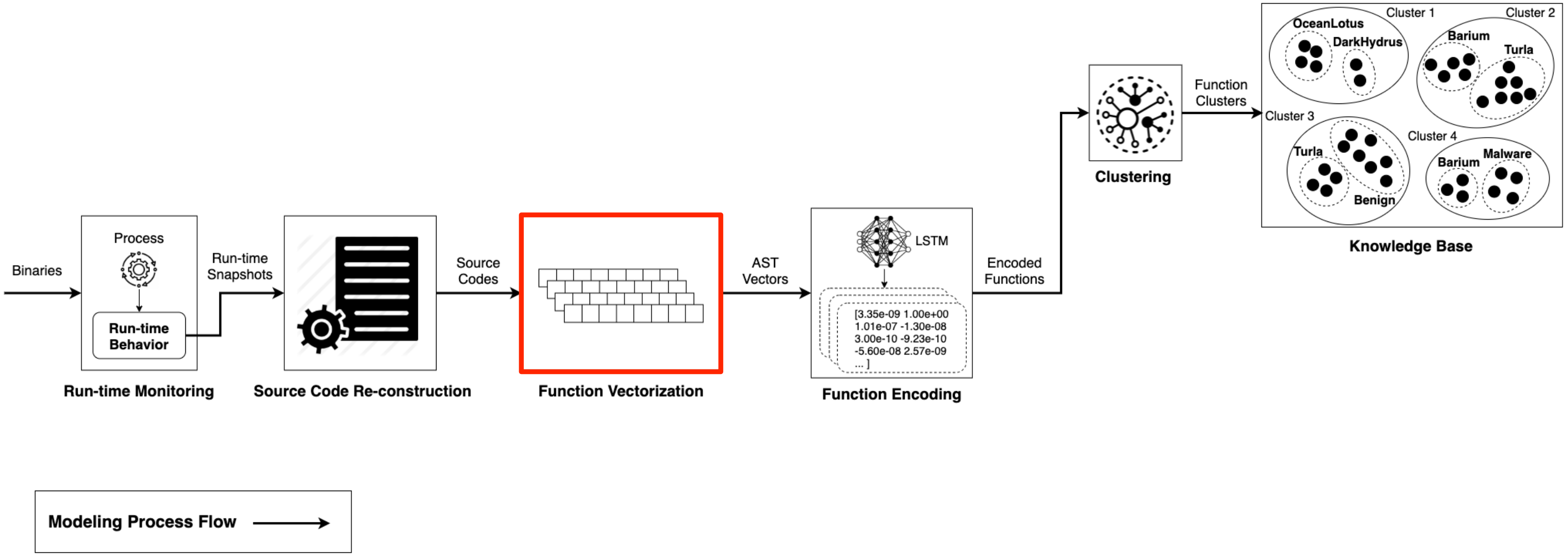
Run-time Monitoring

- *Input*:
  - Malware and benign binaries
- *Output*:
  - Decompiled code

- Steps:
  - Running samples in a dynamic analysis engine
  - Taking snapshots at different stages of the dynamic analysis
  - Re-constructing source code from binaries by integrating decompiled codes of snapshots

Scrutinizer Overview

General Architecture



Function Vectorization

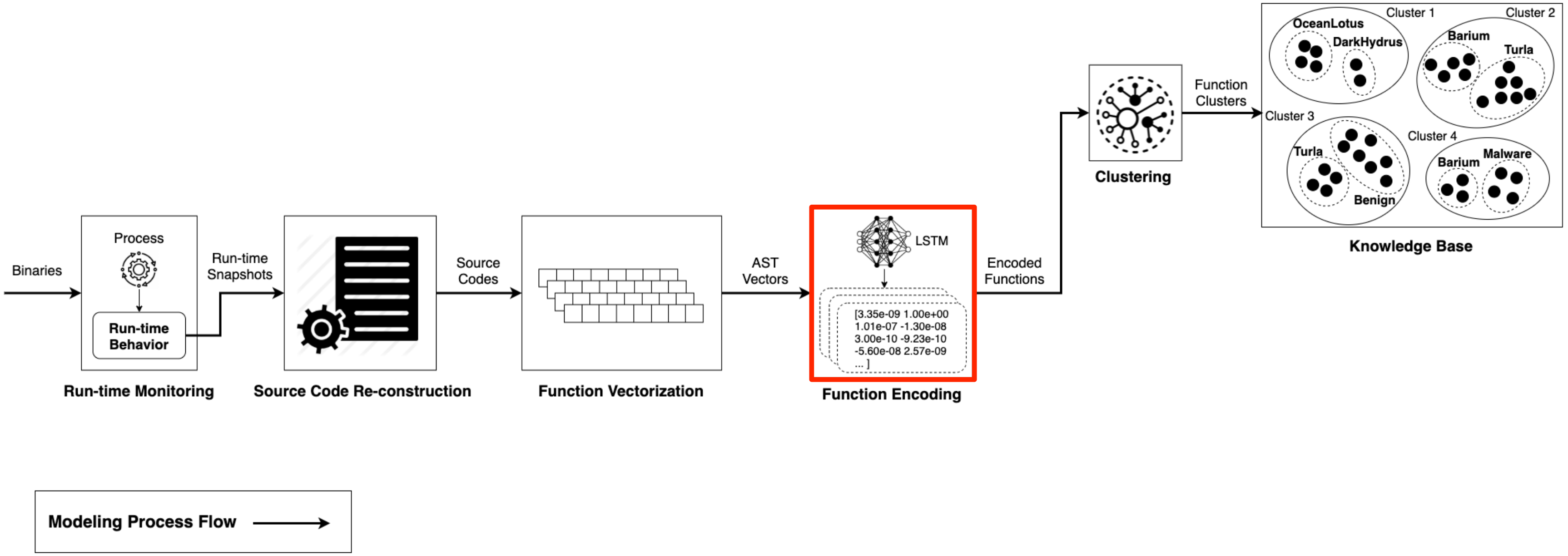
- *Input*:
  - Decompiled code
- *Output*:
  - Abstract Syntax Tree (AST) vector

```
void FUN_1001eab0(void)
{
  ...
  if (pcVar1 == (char *)0x0) {
    pcVar1 = &DAT_10055b20;
  }
  else {
    pcVar1 = pcVar1 + 1;
  }
  wsprintfA(&local_11c,&DAT_10042bf4,pcVar1);
  ...
  LVar3 = RegCreateKeyExA(...);
  if (LVar3 == 0) {
    RegSetValueExA(...);
    RegCloseKey(local_18);
   ...
  }
  ...
  return;
}
```

$$\boldsymbol{V_F} = < \text{FUNCTION\_DECL, DECL\_STMT, VAR\_DECL,}$$
$$\text{DECL\_STMT}, \ldots, \text{IF\_STMT, BINARY\_OPERATOR,}$$
$$\ldots, \boxed{\text{CALL\_EXPR}}\, \text{DECL\_REF\_EXPR}, \ldots, \text{IF\_STMT,}$$
$$\text{COMPOUND\_STMT}, \boxed{\text{CALL\_EXPR}} \ldots, \text{RETURN\_SMT} >$$

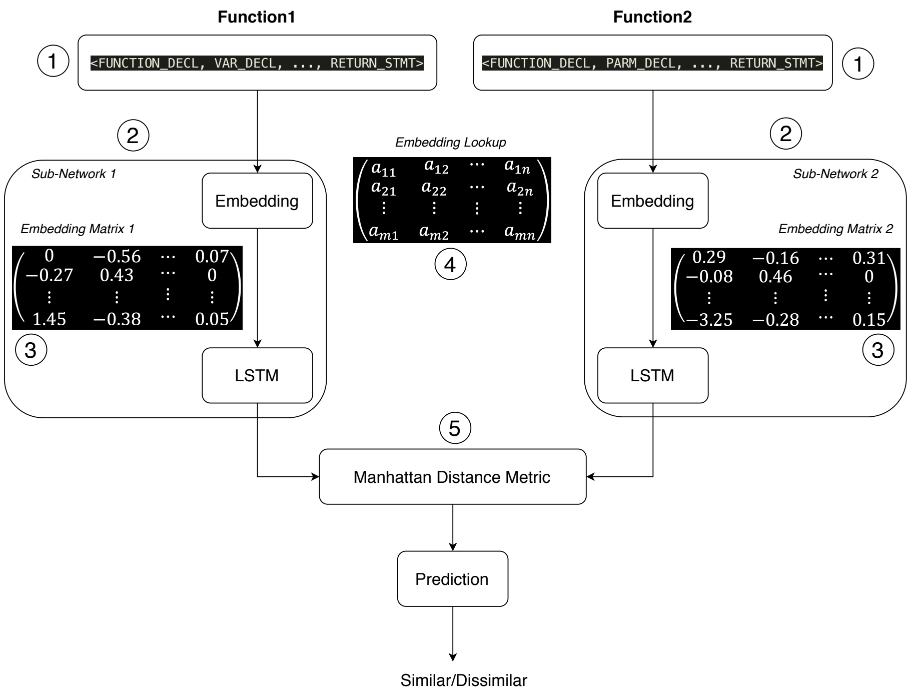
Scrutinizer Overview

General Architecture



Function Encoding

- *Input*:
  - AST vector
- *Output*:
  - Function encoding

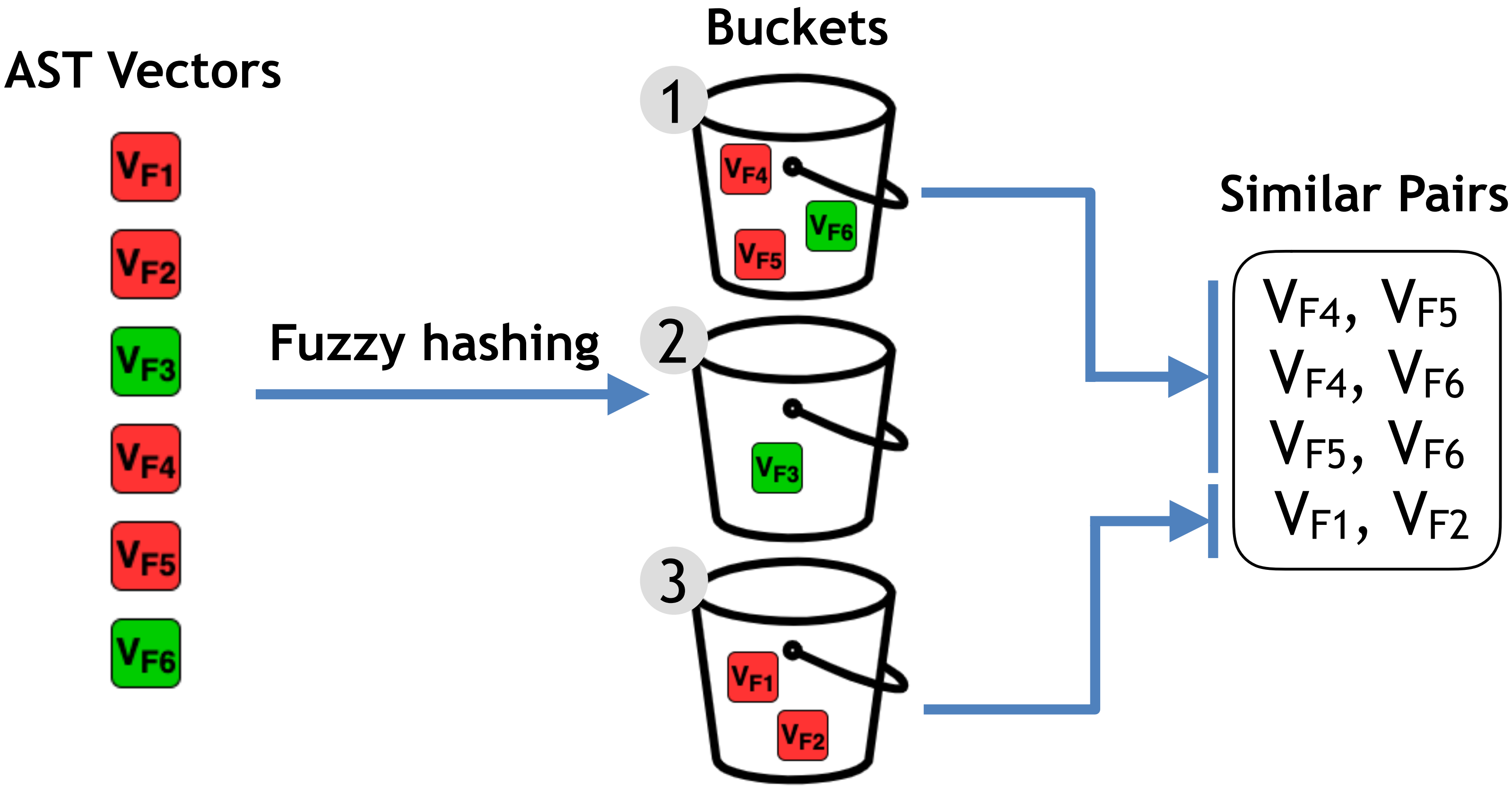


$\mathbf{V_F}$ = <FUNCTION_DECL, DECL_STMT, VAR_DECL, DECL_STMT, ..., IF_STMT, BINARY_OPERATOR, ..., wsprintfA, DECL_REF_EXPR, ..., IF_STMT, COMPOUND_STMT, RegCloseKey, ..., RETURN_STMT>

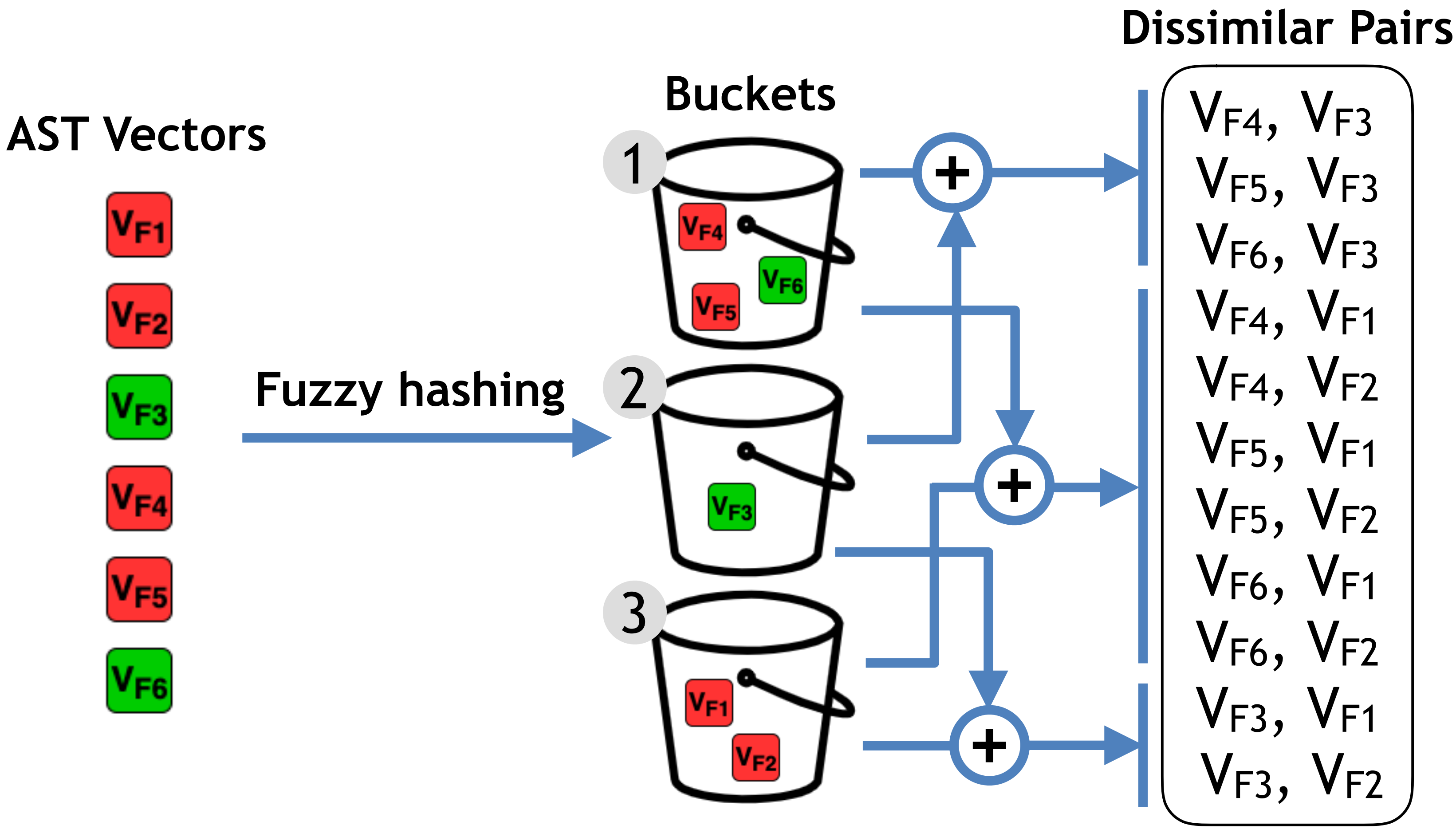
$\mathbf{E_F}$ = <0.29, -0.16, 0.001, 0.23, ..., -0.28, 0.15, 0.08, -0.23, ..., 0.003>

Length = 128

Function Encoding

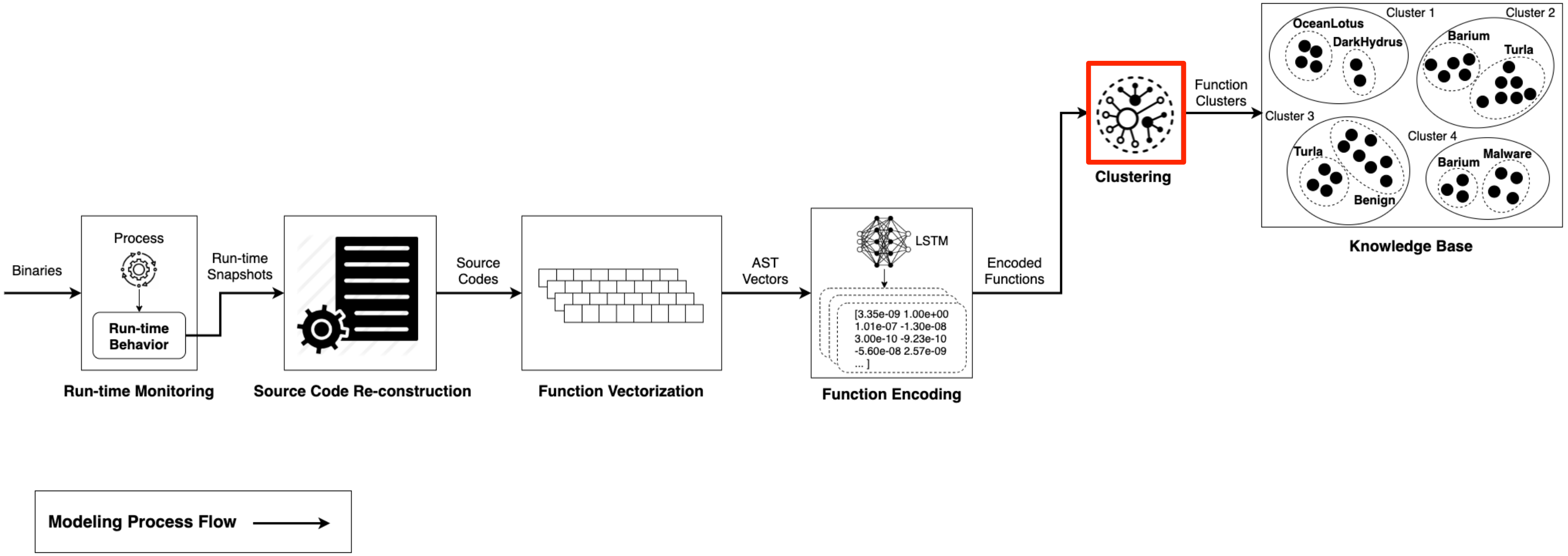


Function Encoding



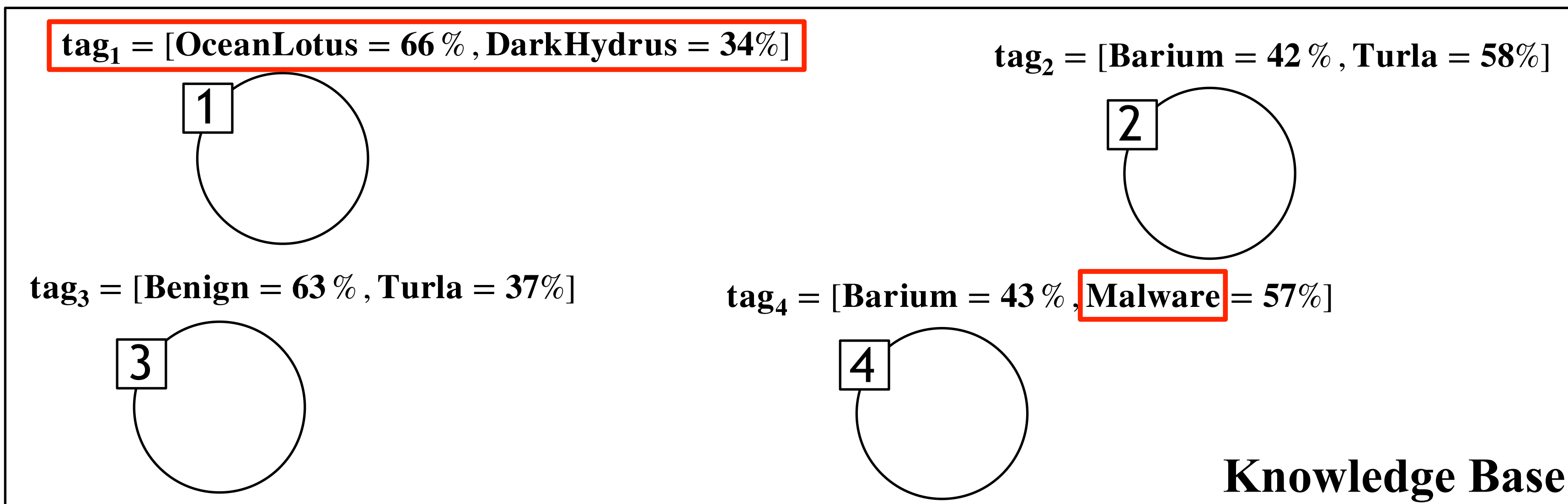
Scrutinizer Overview

General Architecture



Encoding Clustering

- *Input*:
  - Function encodings
- *Output*:
  - Clusters of similar function encodings (knowledge base)

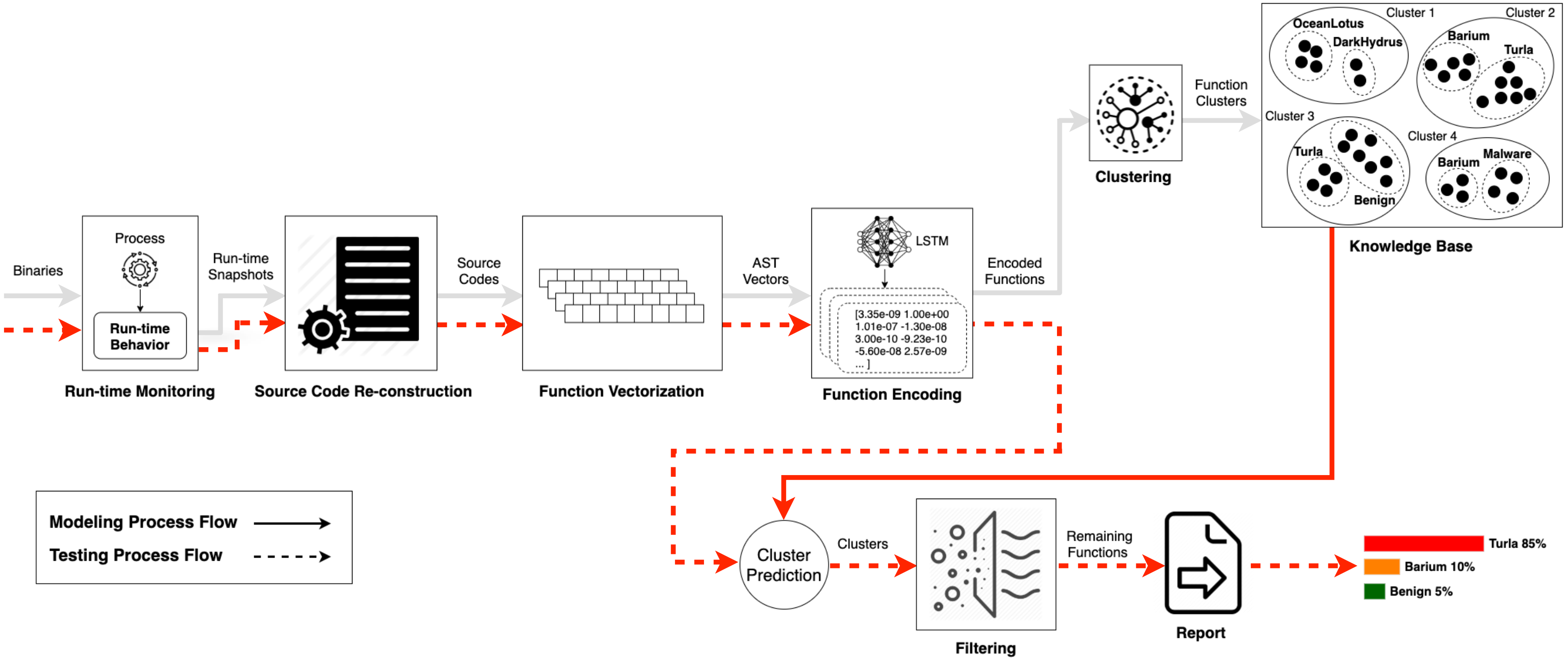


Main Idea

- Identifying code similarities that exist between an unknown sample and those that are known to be used by threat actors from different campaigns
- **Modeling phase**
  - Aim: creating a large knowledge base of previously observed and tagged malware campaigns
- **Testing phase**
  - Aims:
    - Filtering noisy functions
    - Detecting code reuse

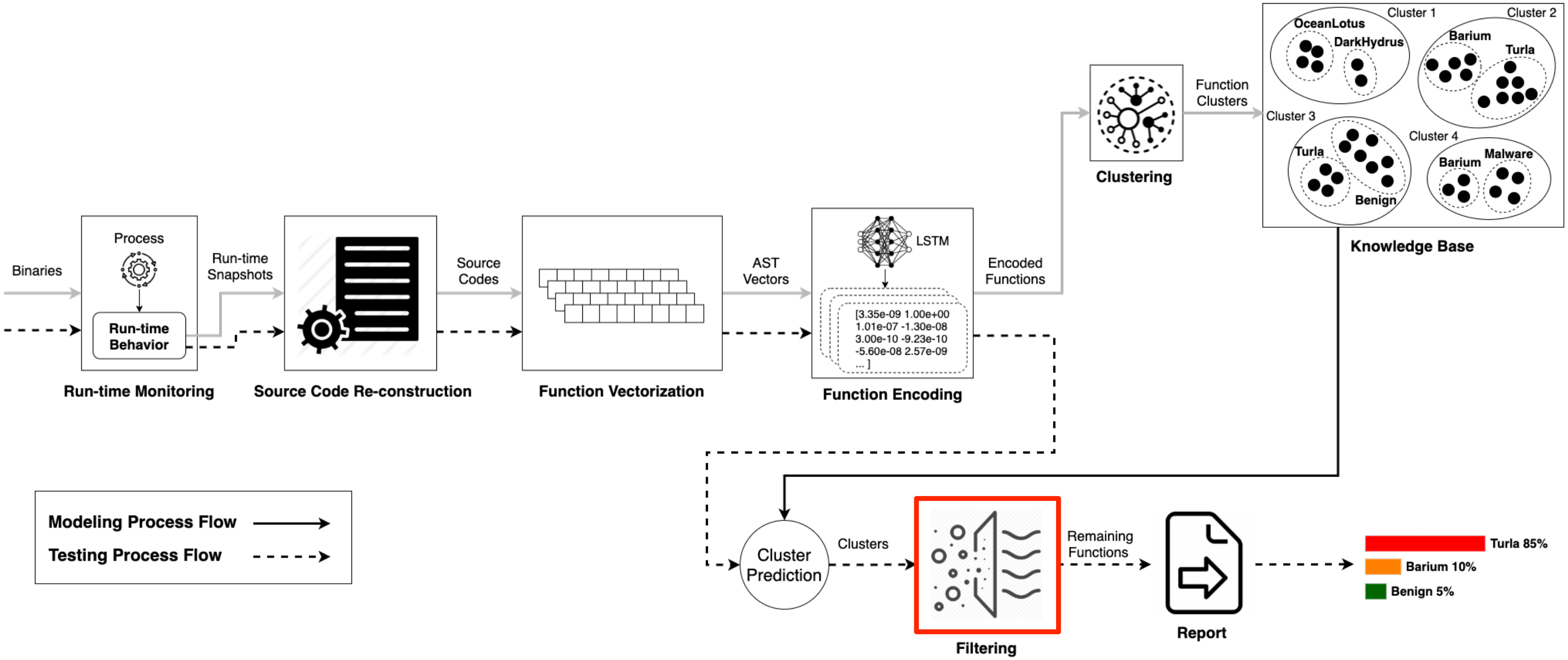
Scrutinizer Overview

General Architecture



Scrutinizer Overview

General Architecture



Filtering Noisy Functions

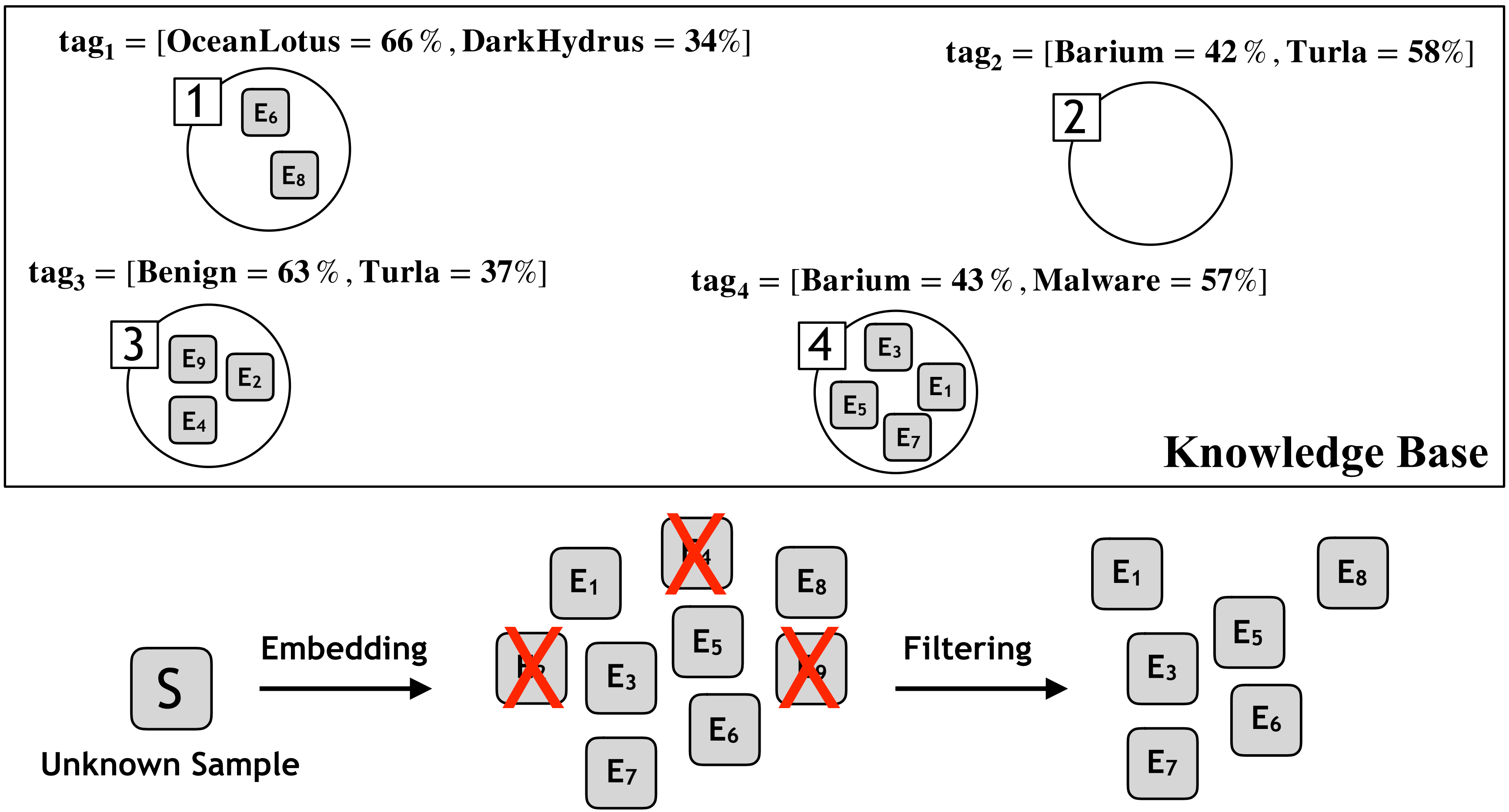
- *Input*:
  - Function encodings
- *Output*:
  - All functions in an unknown sample that are not identified as noisy
  - In other words, functions that are mainly observed in malware

- What are noisy functions and why should they be discarded?
  - Functions that are frequent in both malware and benign samples
  - Malware and benign samples share significant volumes of standard code
  - Shared functions can impact the performance of ML-based systems
  - Analyzing less functions saves resources

Filtering Noisy Functions

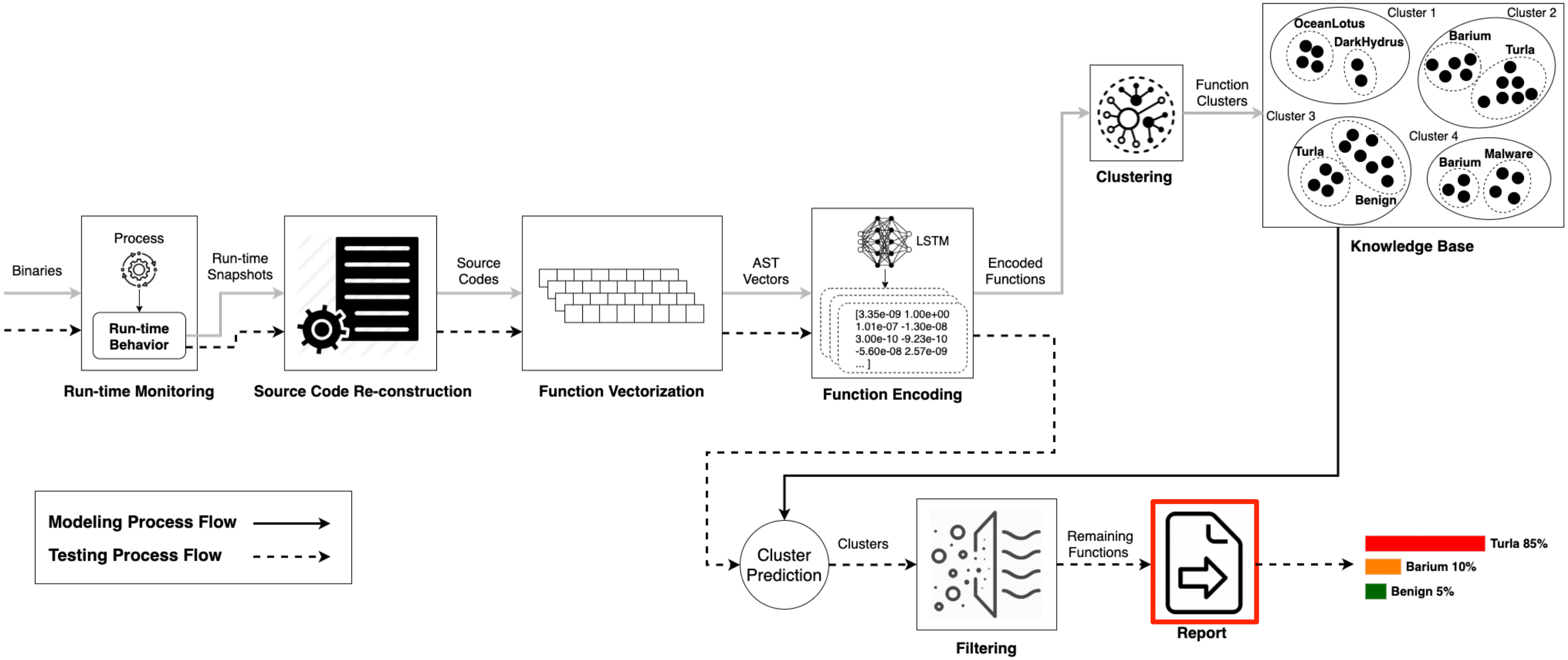
- How noisy functions in an unknown sample are filtered?
  - All functions are encoded initially
  - All functions are assigned to previously known clusters
  - For each function:
    - We first inspect the tag of the cluster to which the function has been assigned
    - If the majority (δ) of functions in the cluster are benign:
      - The function is discarded
    - Otherwise:
      - It is saved for code reuse detection

Filtering Noisy Functions



Scrutinizer Overview

General Architecture

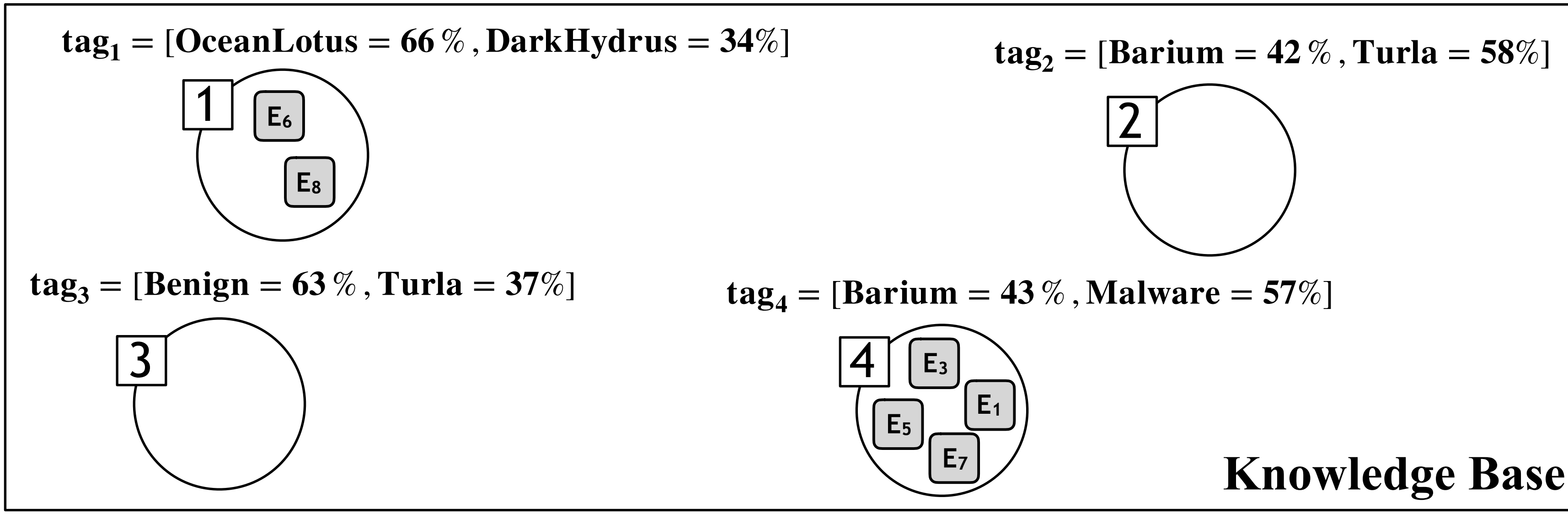


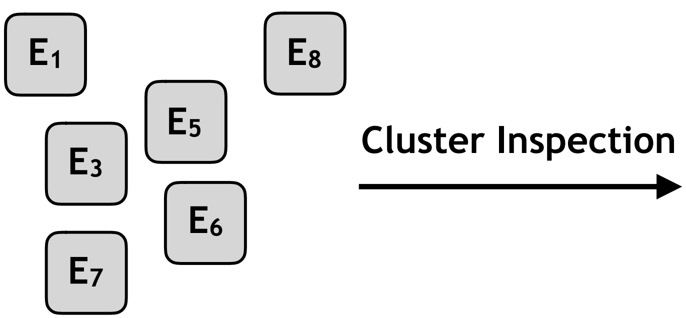
Detecting Code Reuse

- *Input*:
  - Remaining functions from filtering step
- *Output*:
  - A report which shows how much overlap exists between an unknown sample and those which are known to be used by specific campaigns

- How this overlap is detected?
  - Function encodings are assigned to previously created clusters
  - Clusters are inspected automatically to find commonalities

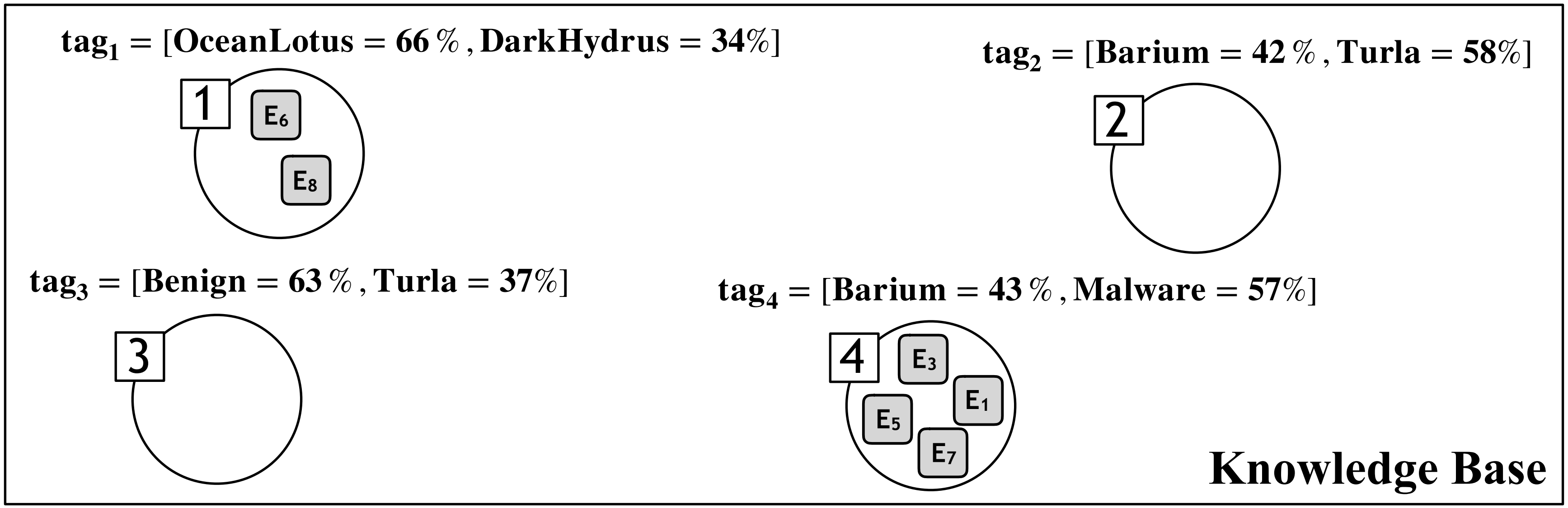
Detecting Code Reuse

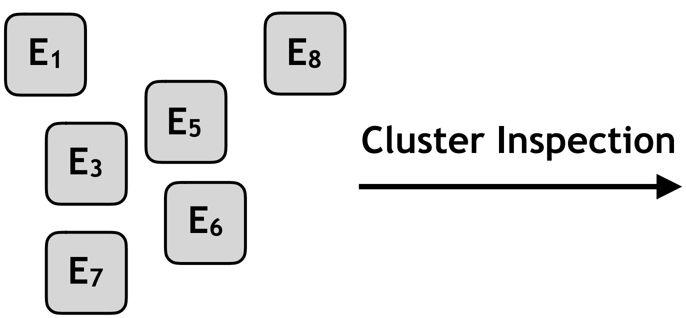




$$c_{OceanLotus} = 1$$
$$c_{DarkHydrus} = 1$$

Detecting Code Reuse

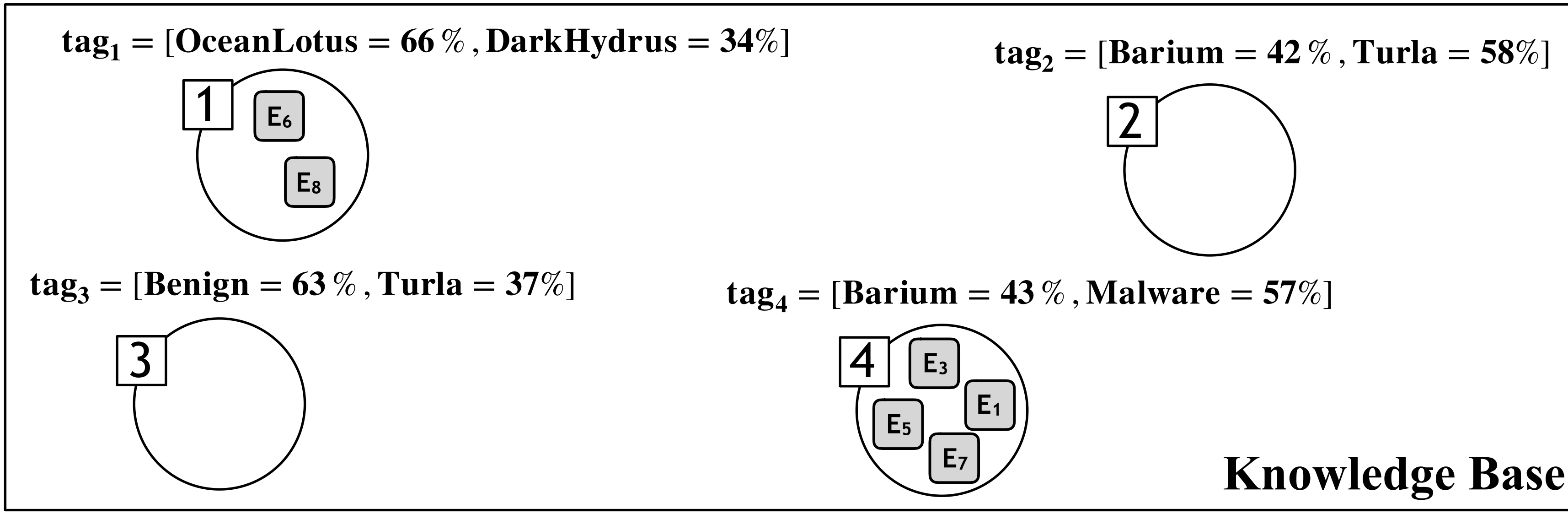


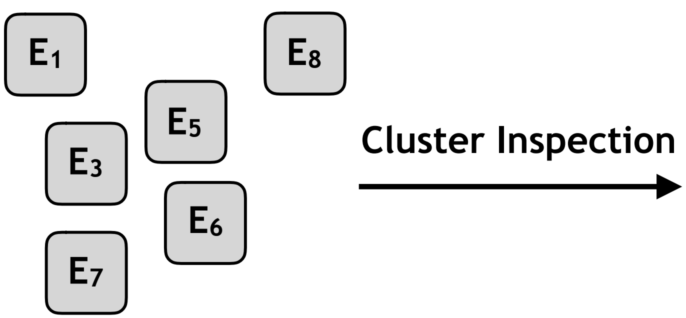


$$c_{OceanLotus} = 2$$

$$c_{DarkHydrus} = 2$$

Detecting Code Reuse





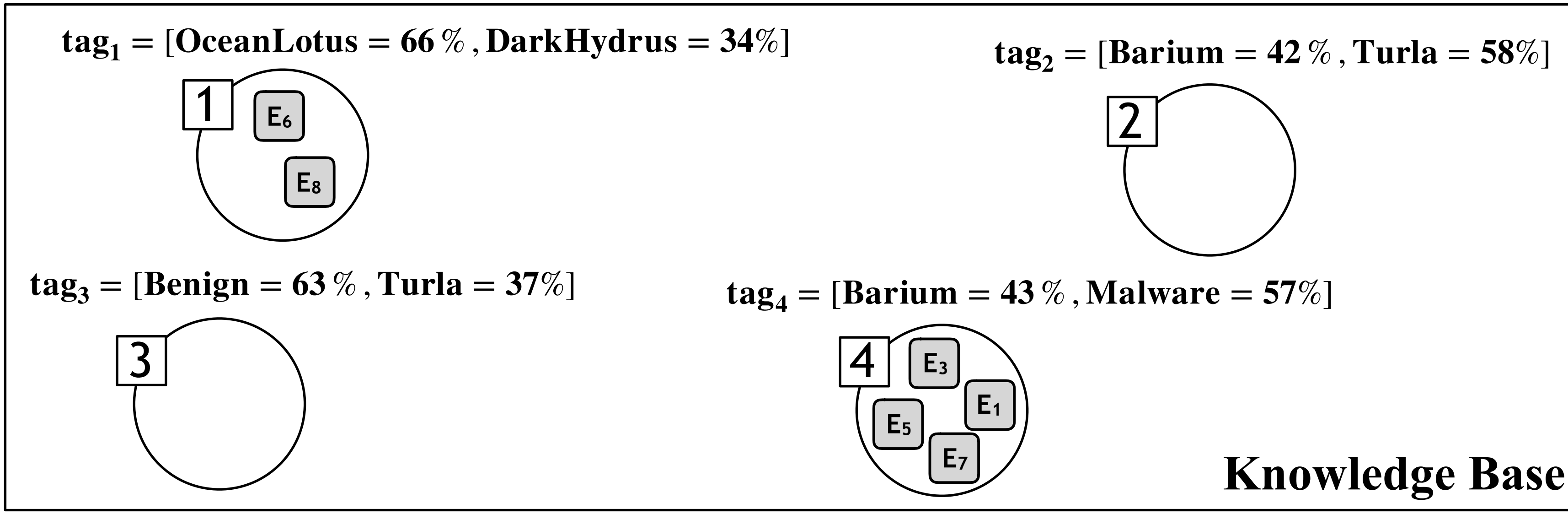
$$c_{Barium} = 1$$

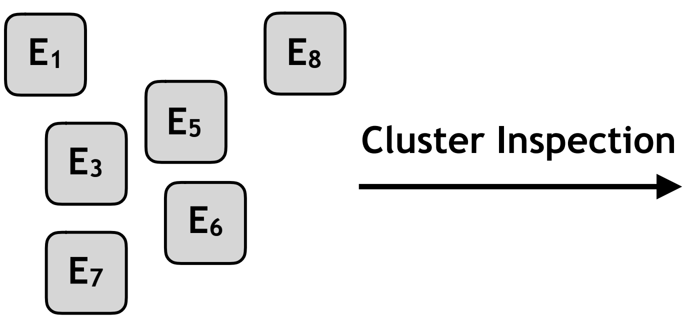
$$c_{OceanLotus} = 2$$

$$c_{DarkHydrus} = 2$$

$$c_{Malware} = 1$$

Detecting Code Reuse





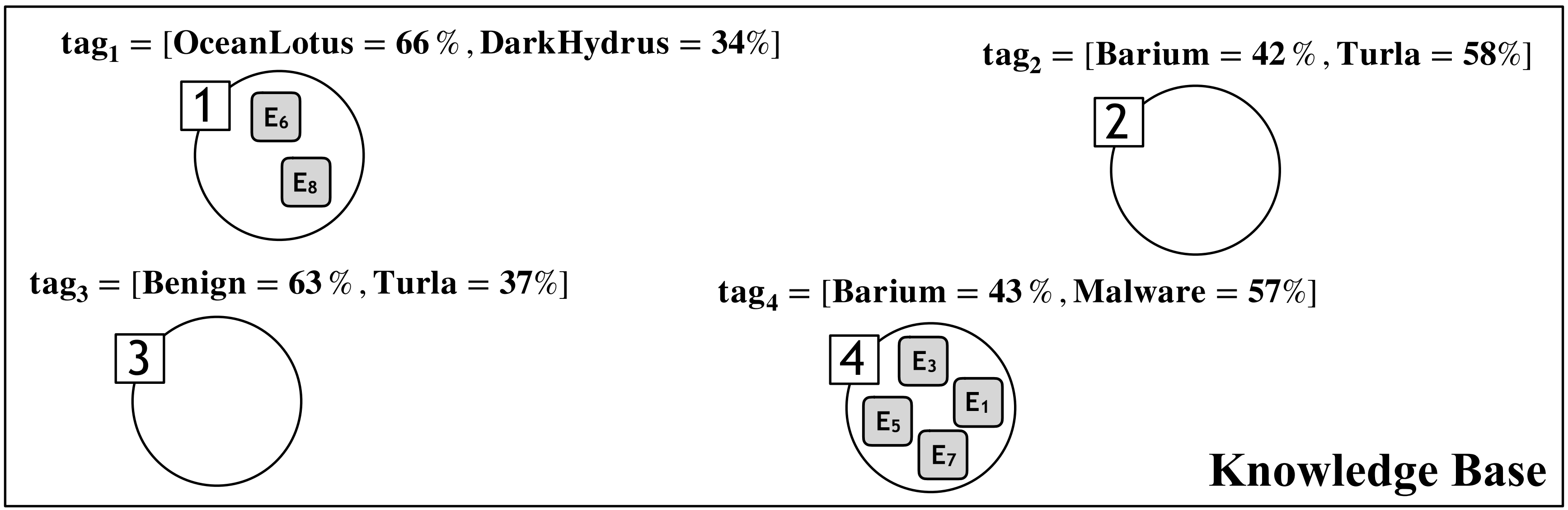
$$c_{Barium} = 2$$

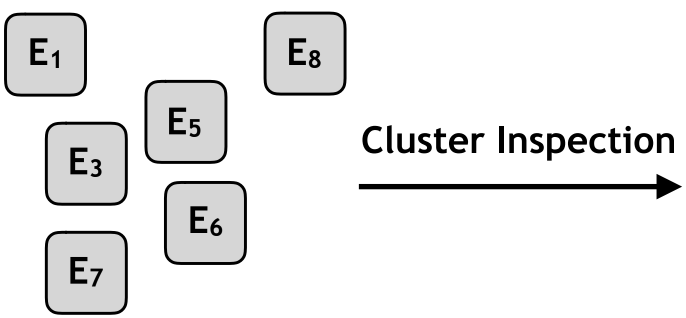
$$c_{OceanLotus} = 2$$

$$c_{DarkHydrus} = 2$$

$$c_{Malware} = 2$$

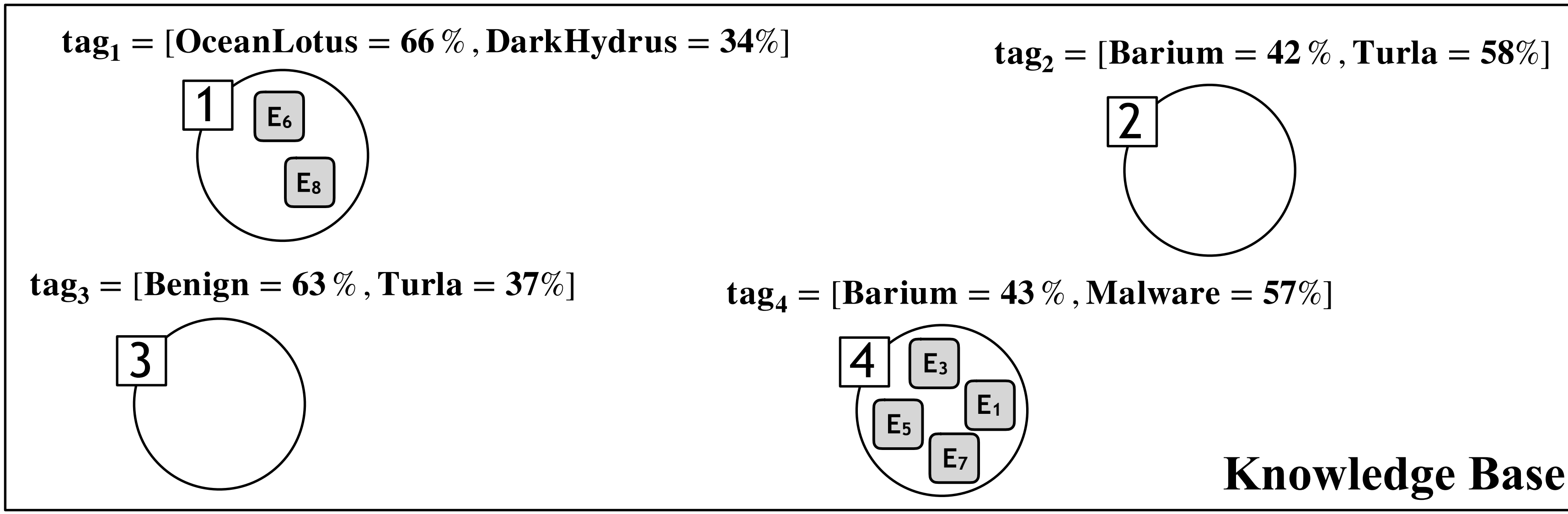
Detecting Code Reuse

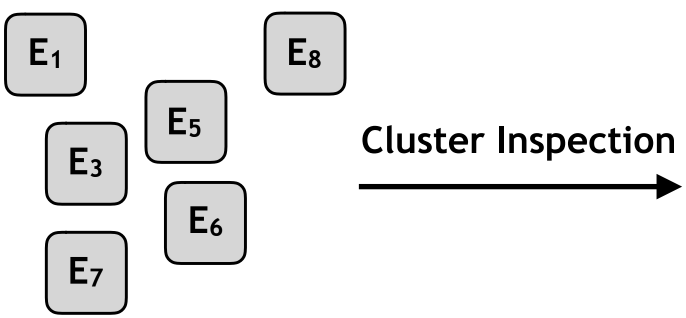




$$c_{Barium} = 3$$
$$c_{OceanLotus} = 2$$
$$c_{DarkHydrus} = 2$$
$$c_{Malware} = 3$$

Detecting Code Reuse





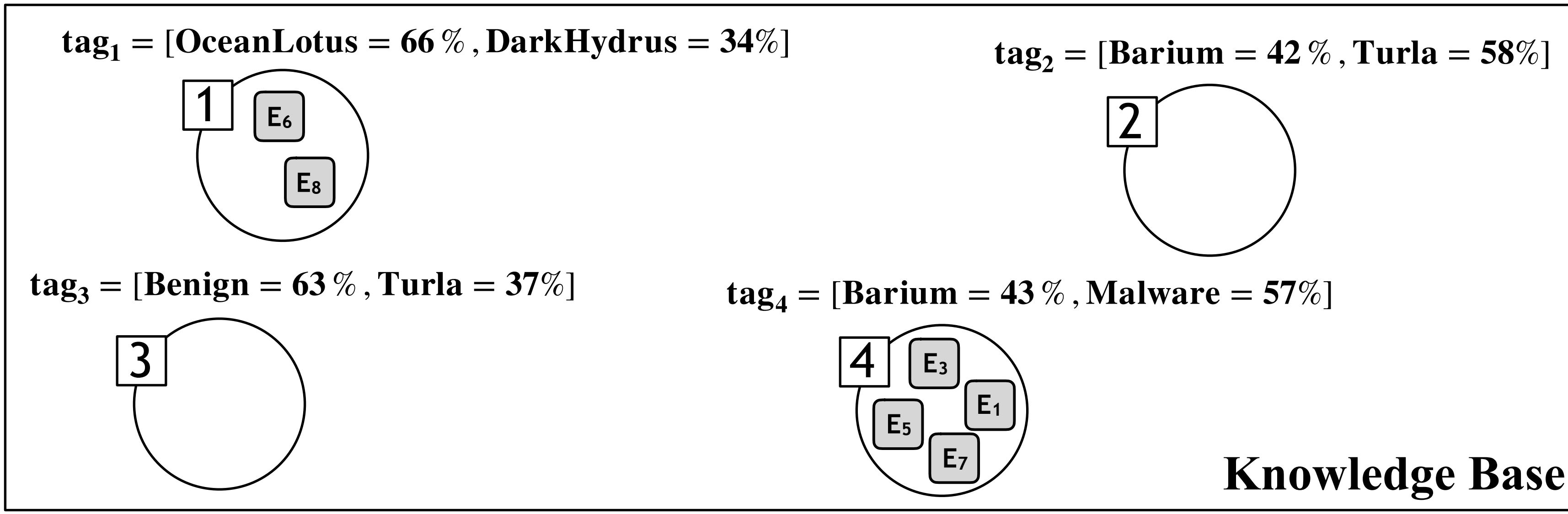
$$c_{Barium} = 4$$

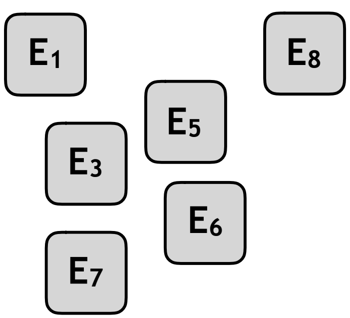
$$c_{OceanLotus} = 2$$

$$c_{DarkHydrus} = 2$$

$$c_{Malware} = 4$$

Detecting Code Reuse





Cluster Inspection →

$$c_{Barium} = 4$$
$$c_{OceanLotus} = 2$$
$$c_{DarkHydrus} = 2$$
$$c_{Malware} = 4$$

Report (x/6) →

$$Barium = 17\,\%$$
$$OceanLotus = 33\,\%$$
$$DarkHydrus = 33\,\%$$
$$Malware = 17\,\%$$

Outline

- Scrutinizer Overview
- **Results**
- Discussion
- Conclusion

Results
Datasets

Phase	Data Type	#Samples	Size	Avg_LOC	Complexity
Modeling	Malware [18]	12,540	0.55	106.21	11.05
	Benign [9]	31,475	0.31	35.73	5.80
	Total	44,015			
Testing	Malware [18]	500	0.38	95.47	10.21
	Benign [18]	2,500	0.29	33.25	5.76
	Total	3,000			

Results
Function Encoding

- Automatic Verification
  - Cross-validation

Prediction error statistics after 5-fold cross-validation

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

- Manual Verification
  - 1000 samples

Results

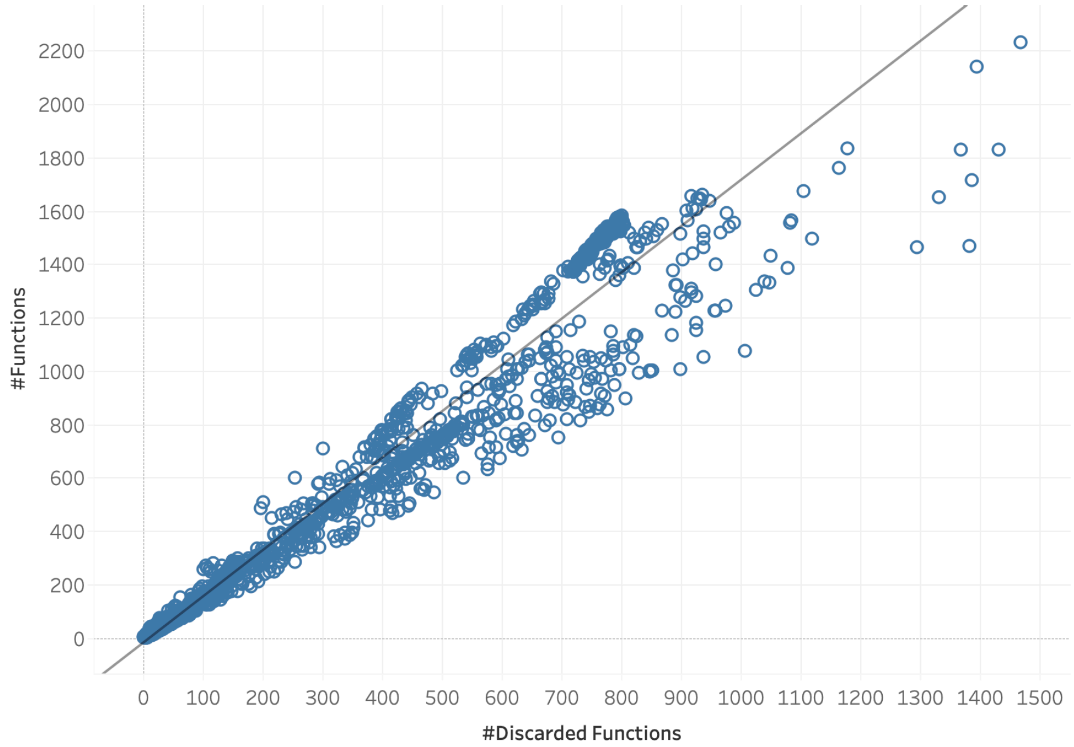
Cluster Analysis

- We leveraged HDBSCAN algorithm to group function embeddings into different clusters
- We reduced the dimension of function embedding from 128 to 8 using PCA to speed up the clustering process
- We could find 1+ million clusters with similar function encodings
  - 91% of clusters were completely benign
  - 3.2% of clusters were completely malicious
  - 5.88% of clusters were mixed
- The average size of clusters was around 5
- The largest cluster had 14K+ function embeddings

Results

Real-World Deployment - Filtering

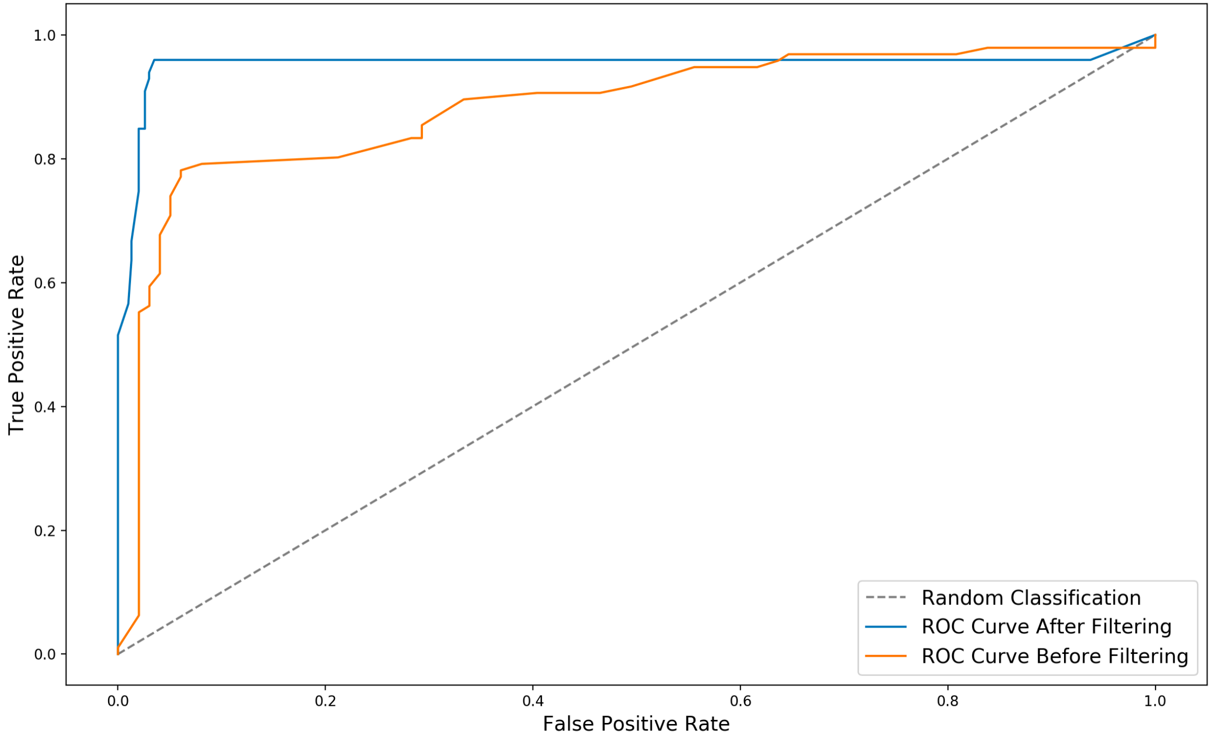
- The filtering mechanism works well in practice by filtering a median of 126 functions (≈56% of code).



Results

Real-World Deployment - Filtering

- The applied filtering mechanism improves the TPR of a classification system by 10% and decreases the FPR by 8.8%



Results

Real-World Deployment - Code Reuse Analysis on APT Campaigns

- Intra-campaign code reuse analysis
- Inter-campaign code reuse analysis

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 [26]
0d67422ba42d4a548e807b0298e372c7	225	55.1	GazaCybergang: 73.9	GazaCybergang [3]
655f56f880655198962ca8dd746431e8	188	66.5	GazaCybergang: 64.0	GazaCybergang [3]
ff8d92dfbcda572ef97c142017eec658	144	70.1	Barium: 38.5	Barium [26][8]
c11dd805de683822bf4922aecb9bfef5	220	65.9	Barium: 38.4	Barium [26][8]
aae531a922d9cca9ddca3d98be09f9df	558	61.6	OilRig: 43.7	OilRig [26][8]
6a7bff614a1c2fd2901a5bd1d878be59	588	59.0	OilRig: 40.6	OilRig [26][8]
a921aa35deedf09fabee767824fd8f7e	44	68.2	GazaCybergang: 41.5	GazaCybergang [26][8]
0e441602449856e57d1105496023f458	73	61.6	Turla: 35.3	Turla [26]
7f05d410dc0d1b0e7a3fcc6cdda7a2ff	220	65.9	Barium: 38.4	Barium [26][8]
557ff68798c71652db8a85596a4bab72	144	70.1	Barium: 38.5	Barium [26][8]
b0877494d36fab1f9f4219c3defbfb19	144	70.1	Barium: 38.5	Barium [26][8]

Outline

- Scrutinizer Overview
- Results
- **Discussion**
- Conclusion

Discussion

- Accuracy
  - Function encoding relies on training data
  - Collecting data is a non-trivial task
    - Decompilation is an error-prone process
    - Features extraction tools cannot handle decompiled codes well due to artifacts
- Analysis costs and potential bottlenecks
  - Dynamic analysis
  - Training and clustering processes

Outline

- Scrutinizer Overview
- Results
- Discussion
- **Conclusion**

Conclusion

- Targeted attacks are growing in number
- Lack of automated tools to inspect code reuse in malware samples that are used in targeted attacks
- We have proposed an automated tool to fill this gap with the following features:
  - An ML-based function encoding mechanism
  - A filtering mechanism to discard functions that are prevalent in both malware and benign samples and to save analysis time
  - An automatic code reuse detection and campaign assignment tool