Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code

December 10-12, 2024, Istanbul, Turkey

Roxane Cohen <rcohen@quarkslab.com>, Quarkslab & LAMSADE, CNRS
Robin David <rdavid@quarkslab.com>, Quarkslab
Florian Yger <florian.yger@dauphine.psl.eu>, INSA Rouen
Fabrice Rossi <fabrice.rossi@dauphine.psl.eu>, CEREMADE

Background: Compilation

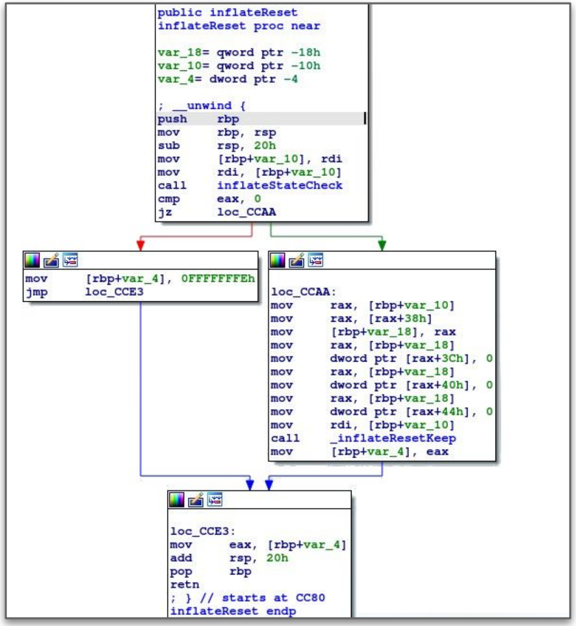
Source Code
(C, Java, Rust)

```c
int ZEXPORT inflateReset(strm)
z_streamp strm;
{
    struct inflate_state FAR *state;

    if (strm == Z_NULL || strm->state == Z_NULL)
    state = (struct inflate_state FAR *)strm->state;
    state->wsize = 0;
    state->whave = 0;
    state->wnext = 0;
    return inflateResetKeep(strm);
}
```

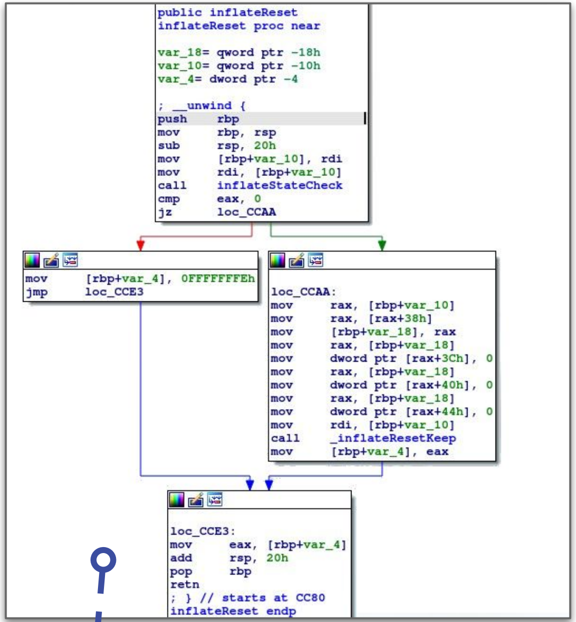
Compilation
(gcc, clang)

Machine Code
(x86, ARM, Aarch64)



Background: Reverse Engineering

Machine Code
(x86, ARM, Aarch64)



Control Flow Graph (CFG)
(Encode loop, and branching condition logic. One for each function)

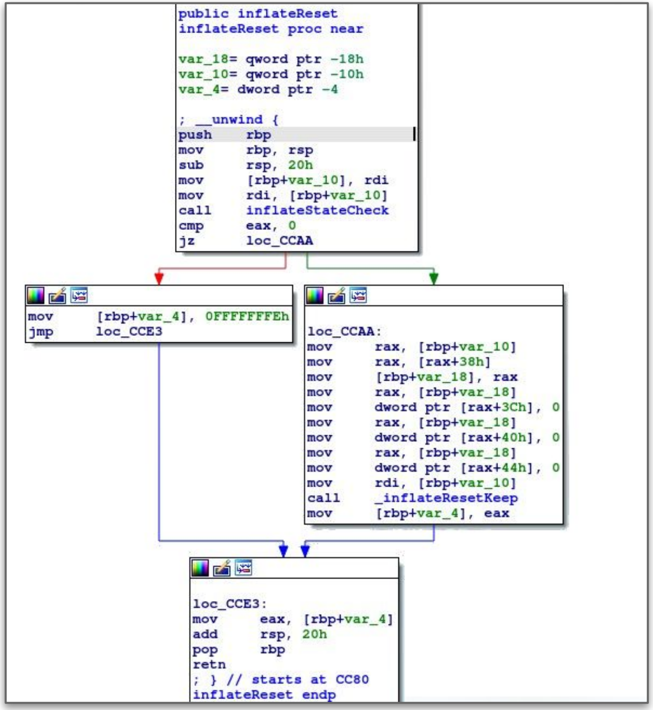
Reverse Engineering

?

Reverse Goal

- ➢ What is the function doing ?
- ➢ Is it legit or suspicious? *(backdoor, malware)*

Background: Reverse Engineering



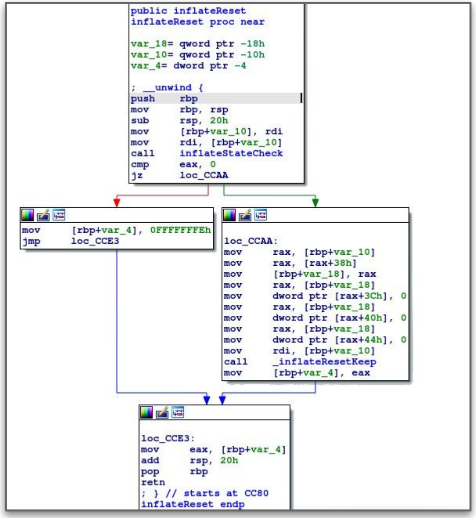
TO DELETE

reverse engineering

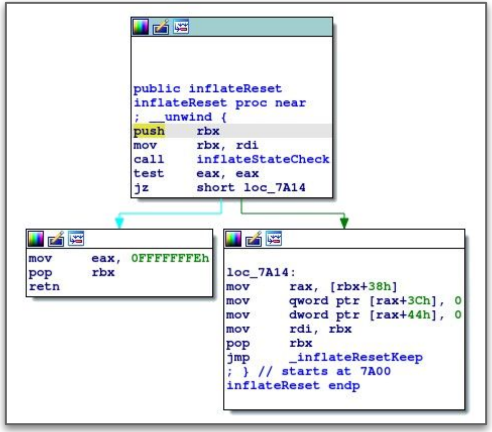
What does this Control-Flow Graph (CFG) do ?

- ➢ Is it related to computation ?
- ➢ Is it related to data ?
- ➢ It is suspicious *(backdoor)* ? Malware-related ?

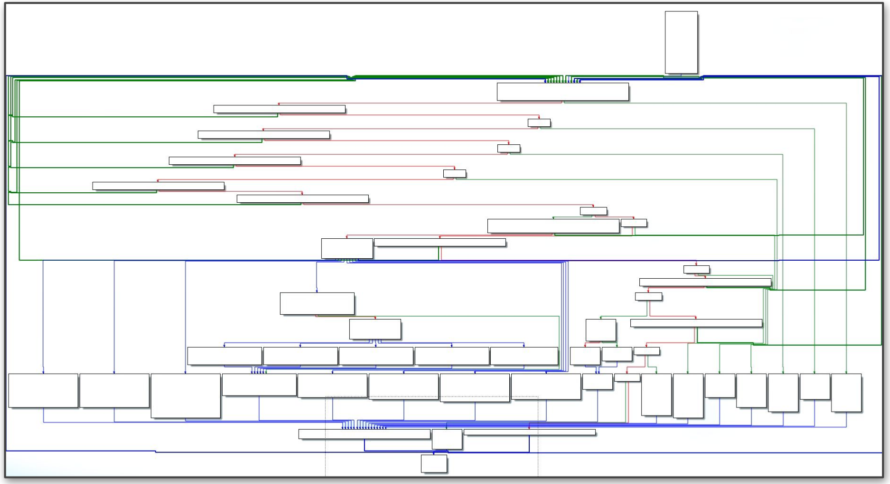
Background: Semanticinary code is not easy



-O0 Compilation
(no optimization)

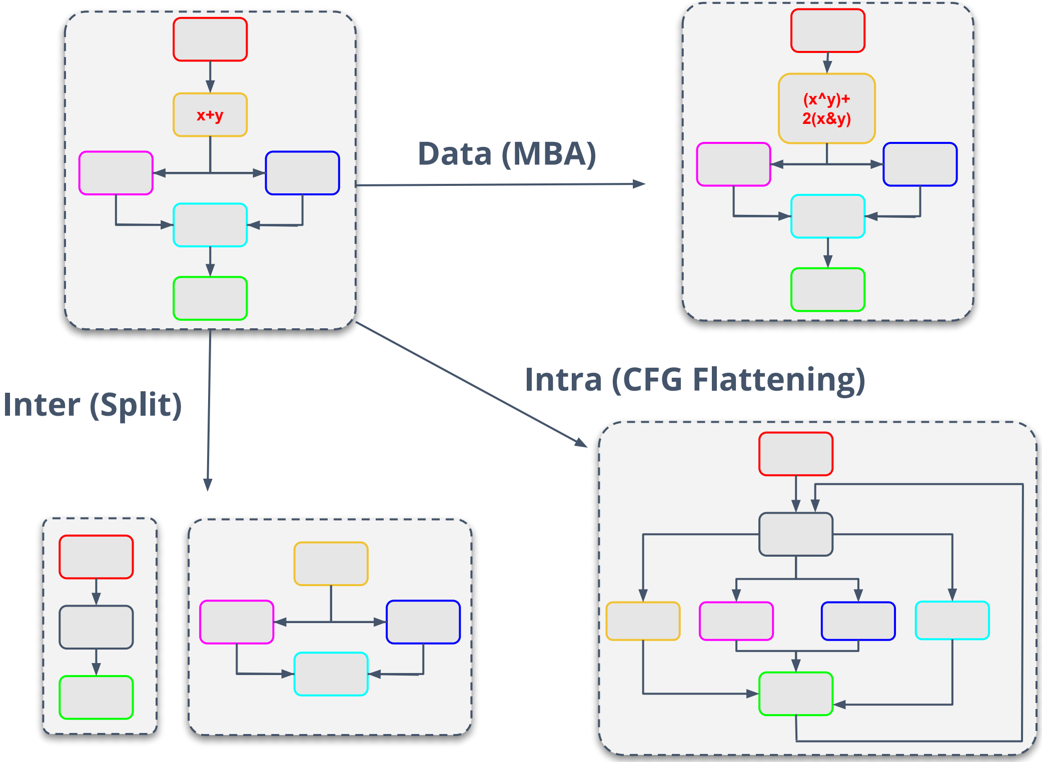


-O2 Compilation
(optimization)



Obfuscation
(virtualization)

Obfuscation



Definition

All the techniques used to alter the syntactic properties of a program without modifying its semantics *(preserving soundness)*

Obfuscation types

- Inter-procedural *(between functions)*
- Intra-procedural *(inside functions)*
- Data *(constants, strings, etc.)*

Obfuscation : reverser's nightmare

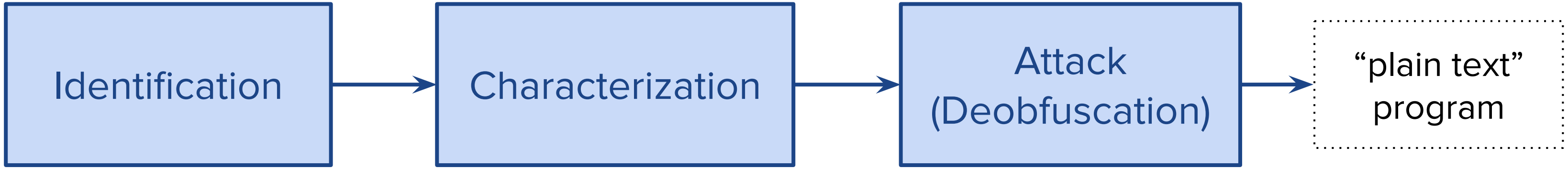
Why ?

Intellectual property *(video games, applications…)*
Malwares *(APT…)*
Diversification

Reverser point of view

Understand what the obfuscation is supposed to hide
Get rid of it *(deobfuscation)*

Step by step obfuscation analysis

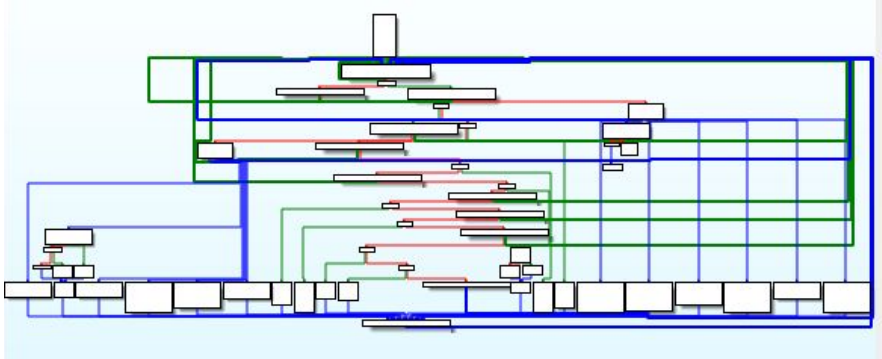


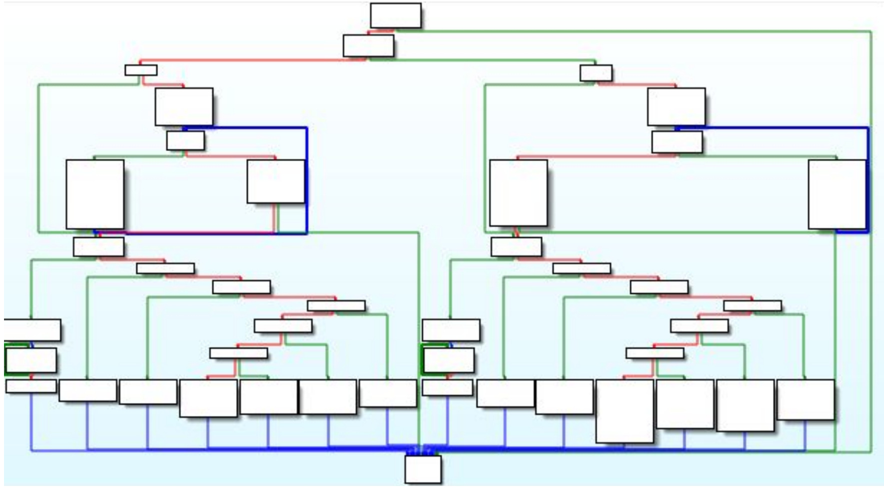
Determining if a binary / function has been obfuscated
⇒ Binary classification

Determining how it was obfuscated
⇒ Multi-class classification

Attack the obfuscation depending on its type
(academic papers start here)

How can we recognize an obfuscated function ?





Which function is obfuscated ? How it is obfuscated ?

Machine Learning for obfuscation detection

Current state-of-the-art

- Little study about classical ML for obfuscation detection [1, 2, 3]
- Little or no study on deep-learning potential for obfuscation detection [4]
- No satisfactory obfuscated dataset available *(too small, not enough obfuscations…)*

Goal : general study about ML for obfuscation analysis

- Evaluating 1) Graph representation 2) Features 3) Models 4) Data in the context of **function obfuscation detection**
- Binary classification vs multi-class classification (11 classes !)

[1] Greco and al.**Explaining binary obfuscation** 2023
[2] Schrittwieser and al. **Modeling obfuscation stealth through code complexity.** 2023
[3] Salem and al. **Metadata recovery from obfuscated programs using machine learning.** 2016
[4] Jiang and al. **Function-level obfuscation detection method based on graph convolutional networks.** 2021

Dataset

Dataset

- **projects**: zlib, lz4, minilua, sqlite, freetype
- **obfuscator**: OLLVM, Tigress
- **obfuscations**:
  - intra *(CFF, Opaque, Virtualization)*
  - inter *(Split, Merge, Copy)*
  - data *(EncodeArithmetic, EncodeLiterals)*
  - mix1 *(intra & data)*
  - mix2 (intra & inter & data)
- High class unbalance

Dataset-1

- Split per function
- Randomly assign functions *(and their obfuscations variants)* to a set *(training, validation, testing)*
- "Easy" setup as two functions belonging to the same program may be close

Dataset-2

- Split per binary
- Assign all the functions of zlib/lz4/minilua *(and their obfuscations variants)* to the training set, sqlite/freetype to the validation/test set
- "Harder" setup: it must generalize to completely unseen binaries

Classical ML

Reminder

- ➤ 1 function = 1 CFG = 1 graph
- ➤ Classical ML : 1 graph = 1 feature vector *(1, d)*

Models

Features	Random Forest	GradientBoosting
Graph-based features	Extract various graph features *(#nodes, #edges, cyclomatic complexity, density)*	
Assembly mnemonic TF-IDF	Use the TF-IDF feature of the 128-most frequent mnemonics inside the function assembly	

Cyclomatic Complexity = E - N + 2P

measure of the complexity of a code (# linearly independent paths)

```
mov eax, 0
```

mnemonic **operands**

Graph Neural Networks

Definition

- ➢ Neural networks adapted to non-euclidean data
- ➢ Invariant to permutation
- ➢ Iteratively update initial node feature given the node neighborhood

$$a_v^{(k)} = AGGREGATE^{(k)}\left(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}\right)$$

$$h_v^{(k)} = COMBINE^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

$$h_G = READOUT(\{h_v^{(K)} | v \in G\})$$

Xu et al. **How powerful are graph neural networks?** International Conference on Learning Representations (2019)

Graph Neural Networks

GCN	$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$	$\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$
SAGE	$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$	
GIN	$\mathbf{x}'_i = h_{\mathbf{\Theta}} \left((1 + \epsilon) \cdot \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j \right)$	
GAT	$\mathbf{x}'_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \mathbf{\Theta}_t \mathbf{x}_j,$	$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^\top \mathbf{\Theta}_s \mathbf{x}_i + \mathbf{a}_t^\top \mathbf{\Theta}_t \mathbf{x}_j\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^\top \mathbf{\Theta}_s \mathbf{x}_i + \mathbf{a}_t^\top \mathbf{\Theta}_t \mathbf{x}_k\right)\right)}$

Comparison of GNN convolution.
GIN offers the best theoretical guarantees *(as powerful as the 1-WL test)*

Graph Neural Networks

Reminder

- 1 function = 1 CFG = 1 graph
- GNN : 1 graph ~ 1 feature vector per node !

Features

- Identity feature *(vector filled with 1's)*
- Coarse assembly feature : counting the number of assembly classes *(floating-point mnemonics, data-transfer mnemonics…)*
- "Semantic" assembly feature : counting the assembly mnemonics *(mov, lea, …)*
- "Semantic" Pcode feature : counting the Pcode mnemonics *(BRANCH, STORE,…)*

Pcode is an intermediary representation that translates an assembly instruction into an architecture-agnostic language

Advantage : only 72 Pcode mnemonics ! Assembly mnemonics > 1800.

Evaluation

How can we compare the functions pair that should be matched ***(Ground-Truth)*** **and the functions that are matched by a differ on stripped binaries ?**

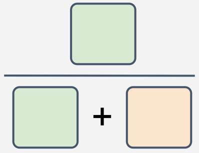
True Positives
good match correctly identified

False Positives
wrong match identified

True Negative
Not a match considered as-is

False Negative
Good match **not** identified

Recall =



⇒

$$\textbf{balanced accuracy} = \frac{\textbf{Recall(c0) + ... + Recall(cn)}}{\textbf{n}}$$

Binary classification

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	***Dataset-2***
CFG	Graph features & assembly (Dim: #**23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: #**128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: #**1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: #**27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: #**78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: #**1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

Binary classification

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	*Dataset-2*
CFG	Graph features & assembly (Dim: **#23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: **#1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: **#27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: **#78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: **#1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

Stable baselines, with better scores using GB and mnemonic TF-IDF

Dataset-1 have higher score than Dataset-2

Binary classification

GNN with coarse features give disappointing results.

Meaningful features *("semantic")* outperform baselines

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	*Dataset-2*
CFG	Graph features & assembly (Dim: **#23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: **#1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: **#27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: **#78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: **#1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

Binary classification

Pcode feature outperforms assembly feature while being less costly (#78 instead of #1839) and CPU-agnostic

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	*Dataset-2*
CFG	Graph features & assembly (Dim: **#23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: **#1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: **#27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: **#78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: **#1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

Binary classification

Graph	Features	Algorithm	Balanced accuracy Dataset-1	Balanced accuracy Dataset-2
CFG	Graph features & assembly (Dim: #**23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: #**128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: #**1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: #**27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: #**78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: #**1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

Better generalization capabilities of GNN compared to baselines

Multi-class classification (11 classes)

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	*Dataset-2*
CFG	Graph features & assembly (Dim: **#23**)	RandomForest	0.65	0.57
		GradientBoosting	0.66	0.594
	TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.697	0.593
		GradientBoosting	0.724	0.579
	Identity (Dim: **#1**)	GCN	0.323	0.326
		Sage	0.341	0.347
		GIN	0.414	0.407
		GAT	0.192	0.195
		UNet	0.362	0.299
	Counting mnemonic classes (Dim: **#27**)	GCN	0.431	0.462
		Sage	0.498	0.499
		GIN	0.488	0.474
		GAT	0.45	0.342
		UNet	0.439	0.448
	Semantic & counting PCode mnemonics (Dim: **#78**)	GCN	0.721	0.675
		Sage	0.737	0.549
		GIN	0.732	0.657
		GAT	0.729	0.637
		UNet	0.704	0.655
	Semantic & counting assembly mnemonics (Dim: **#1839**)	GCN	0.723	0.633
		Sage	0.718	0.535
		GIN	0.713	0.427
		GAT	0.723	0.646
		UNet	0.709	0.611

Multi-class classification (11 classes)

Graph	Features	Algorithm	Balanced accuracy	
			Dataset-1	*Dataset-2*
CFG	Graph features & assembly (Dim: **#23**)	RandomForest	0.65	0.57
		GradientBoosting	0.66	0.594
	TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.697	0.593
		GradientBoosting	0.724	0.579
	Identity (Dim: **#1**)	GCN	0.323	0.326
		Sage	0.341	0.347
		GIN	0.414	0.407
		GAT	0.192	0.195
		UNet	0.362	0.299
	Counting mnemonic classes (Dim: **#27**)	GCN	0.431	0.462
		Sage	0.498	0.499
		GIN	0.488	0.474
		GAT	0.45	0.342
		UNet	0.439	0.448
	Semantic & counting PCode mnemonics (Dim: **#78**)	GCN	0.721	0.675
		Sage	0.737	0.549
		GIN	0.732	0.657
		GAT	0.729	0.637
		UNet	0.704	0.655
	Semantic & counting assembly mnemonics (Dim: **#1839**)	GCN	0.723	0.633
		Sage	0.718	0.535
		GIN	0.713	0.427
		GAT	0.723	0.646
		UNet	0.709	0.611

Same trend than in the binary case !

Results are very promising given the high number of classes

Real-world example : XTunnel

XTunnel

- Malware designed by APT28 *(Russia)*
- Used to exfiltrate data from a compromised device
- Obfuscated with OpaquePredicates [1]

	Binary balanced accuracy	Multi-class balanced accuracy
Sample C637E	0.726	0.533
Sample 99B45	0.711	0.55

[1] Bardin and al. **Backward-bounded dse: Targeting infeasibility questions on obfuscated codes**. 2017

Conclusion

Obfuscation detection and classification

- Promising results, with satisfactory baselines
- GNN need meaningful features conveying part of the function "semantics"
- GNN with a strong generalization power
- High results, both for the binary and multi-class classification
- Concrete example with malware obfuscation detection

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: quarkslab.com

@quarkslab

Quarkslab