Aversarial example againt SotA ML-based binary function classifiers

Gabriel Sauger

Loria

8th Franco-Japanese security workshop

Problem statement

Context:

- Target: Static binary **function** classification using machine-learning techniques.
  - Graph-based features
  - Assembly code analysis features (bytegrams, n-grams, strings...)

A classifier claim to extract the *semantics* of a given function, no matter the syntax (optimization / obfuscations).

Problem statement

Question: Can we build an obfuscator to automatically induce a misclassification in to SotA binary function classification models ?

SotA papers in binary function classification

Marcelli et al (*How Machine Learning Is Solving the Binary Function Similarity Problem)* benchmarked several SotA binary function classifiers. We selected the best results:

- *Asm2vec* (assembly level 3-grams - unsupervised)
- Google's *GGSNN and GMN* (graph and assembly level features)

All have near 0.90 on the benchmark performance metrics.

Threat model

- Attacker can use their own compiler toolchain to compile the malicious binary.
- Attacker **does not have access** to $\mathcal{C}$ but knows the *probable* list of features used to construct $\mathcal{C}$. This means that attacker knows the range of features on which the static analysis tools perform their analysis.
- The payload p is "smaller" than the target t
- Defender is restricted to **static analysis** based tools

Notations

- p: *payload* function
- t: *target* function, known to be benign to the defender
- $p_{asm}, p_{src}, p_{bin}$ is the *assembly code, source code, binary* version of p
- $[\![p]\!]$ is the *semantics* of p

Obfuscation recipe

1. **Run the CFG Merger**:

$$(p_{src}, t_{src}) \rightarrow p'_{src} \text{ with } [\![p]\!] = [\![p']\!]$$

Obfuscation recipe

1. **Run the CFG Merger**:

$$(p_{src}, t_{src}) \rightarrow p'_{src} \text{ with } [\![p]\!] = [\![p']\!]$$

2. Compile to assembly code p'_{src} and t_{src}

Obfuscation recipe

1. **Run the CFG Merger**:

$$(p_{src}, t_{src}) \rightarrow p'_{src} \text{ with } [\![p]\!] = [\![p']\!]$$

2. Compile to assembly code p'_{src} and t_{src}
3. Run the **instruction frequency alignement algorithm**

$$(p'_{asm}, t_{asm}) \rightarrow p''_{asm} \text{ with } [\![p]\!] = [\![p'']\!]$$

Obfuscation recipe

1. **Run the CFG Merger**:

$$(p_{src}, t_{src}) \rightarrow p'_{src} \text{ with } [\![p]\!] = [\![p']\!]$$

2. Compile to assembly code p'_{src} and t_{src}
3. Run the **instruction frequency alignement algorithm**

$$(p'_{asm}, t_{asm}) \rightarrow p''_{asm} \text{ with } [\![p]\!] = [\![p'']\!]$$

4. Compile p''_{asm} to obtain p''_{bin} and feed it to the classifier.

CFG Merger: Principle

First step: CFG merge

CFG Merger: Principle

Algorithm 1 Tree Matching algorithm, main function

Require: $(Prog, Prog)$
Ensure: $Prog$ if match is possible, else $None$

```
1: function MERGE_PROG(s, t)
2:     match (s.cf, t.cf) with
3:         | (If, If)
4:             res ← MERGE_PROG(s.cf.body, t.cf.body)
5:             match res with
6:                 | Prog                                   ▷ We matched the bodies
7:                     res2 ← MERGE_PROG(s.cf.tail, t.cf.tail)
8:                     match res2 with
9:                         | Prog                           ▷ And we matched the tails !
10:                            return Prog(s.data, res, res2)   ▷ Amazing, we have a match :)
11:                        | None
12:                            return MERGE_AFTER_INSERT(s,t)  ▷ Here we need to insert a node from t in s
13:                    end match
14:                | None
15:                    return MERGE_AFTER_INSERT(s,t)
16:            end match
17:        | (While, While)
18:            Same as previous case                ▷ Same disjonction as precedently but with two While
19:        | (If, While) or (While, If) or (End, If) or (End, While)
20:            return MERGE_AFTER_INSERT(s,t)
21:        | (End, End)
22:            return Prog(s.data, End)                     ▷ Termination case, match :)
23:        | (_, End)
24:            return None                                  ▷ Termination case, no match :(
25:    end match
26: end function
```

CFG Merger: Principle

Second step: instruction matching

Instruction-level obfuscation problem statement

We target an instruction-level feature based classifier $\mathcal{C}$. Given **the (x86) assembly code files** of a payload p' and a target t, our goal is still to build a program p'' such that:

- $[\![p']\!] = [\![p'']\!]$ (p semantically equivalent to p')
- **Idea:** Aligning the histogram of instructions is enough to fool the classifier. (see Damasio et al *A Game-Based Framework to Compare Program Classifiers and Evaders*)

In this example, the score of $\mathcal{C}$ is equal to 0 when the functions are **dissimilar** and 1 when they are the **same** semantically.

Assembly-level features

The selected features of an assembly **instruction** are:

- its *operation* (the mnemonic)
- its *operands*
  - register name
  - "immediate" if it's an address
  - reference if it's a "[$eax + imm$]"-like reference address

Instructions obfuscation : hand-crafted examples

The steps are:

1. Generate a bunch of $\{(p_i, p'_i, t_i)\}_i$ with the *merger*. $p \in sqlite$ and $t \in curl$.
2. $\forall i$, build the **diff table i** of p'_i and t_i.
3. $\forall i, (p' i_{asm}, \textbf{diff table i}) \rightarrow p''_i$
4. Compute $a2v(p''_i, t_i)$, $a2v(p_i, t_i)$ and $a2v(p''_i, p_i)$. Compare.

Diff Table example

Given a pair of function (p, t), we compute the **difference** between corresponding items in $Features(p)$ and $Features(t)$:

feature	Feat(p,t) $= \#t - \#p$
mox, rdx, rax	8
$callimmediate$	2
...	...
$learax, [rbp + 0x3]$	-5

The *Pair dataset*

An **eligible** function is a function that:

- Has **more** than 5 basic blocks
- Is **parsable** by our CFG Merger

Binary	Nb compilable functions
source:*curl* - target:*openssl*	78
source:*curl* - target:*sqlite3*	216
Total	294

NOTE

curl and *openssl* binaries are included in the classifier's training dataset. *sqlite3* is not.

Benchmarks

Targets:

- GNN
- GMN
- asm2vec

Instruction-level obfuscation: Misclassification problems

When we talk about **misclassification**, we refer to two problems:

"Query" problem

The defender has a database of function. We present him p'. The defender can then query his database, by looking at the top k highest-similarity functions according to $\mathcal{C}$. Our goal then is to have t in the returned functions and not p.
One **metric** commonly used in this case is the **MRR@k** score.

"Triplet" problem

When given a triplet (p, t, p'), we want $\mathcal{C}$ to output that p' looks more similar to t than to p.
Formally, if $\mathcal{C}(p, t)$ is low, our goal is:

$$\mathcal{C}(p', t) > \mathcal{C}(p', p) \text{ and } \mathcal{C}(p', t) > \mathcal{C}(p, t) \quad (1)$$

Results: CFG Merger only

Results on the selected pairs, on the "Triplet problem":

Dataset	Classifier	"Good" triplets ratio
(*curl*,*sqlite*3)	GNN	155/230 **(0.73)**
(*curl*,*openssl*)	GNN	63/78 **(0.81)**
(*curl*,*sqlite*3)	GMN	38/211 **(0.18)**
(*curl*,*openssl*)	GMN	16/78 **(0.21)**
(*curl*,*sqlite*3)	a2v	6/211 **(0.03)**
(*curl*,*openssl*)	a2v	1/78 **(0.01)**

Results: CFG Merger + **random** instruction alignement algorithm

Results on the selected pairs, on the "Triplet problem":

Dataset	Classifier	"Good" triplets ratio
(*sqlite3*,*curl*)	a2v	**(0.75)**

Next steps

While technically enough to have one misclassification with reasonnable "chance":

- Use semantic preserving instruction insertions
- Using save-regs / restore-regs functions
- "hiding" instructions in dead branches
- inserting sets of instructions equivalent to no-op
- compare to the Random Forest of Damasio et al.