StuxNNet:

Practical Live Memory Attacks on Machine Learning Systems

Raphael Norwitz, Bryan Kim

Background

- Prior work:
  - Mostly focused on generating/producing robustness to **adversarial inputs**
  - No one has attempted to modify the model itself
- DNN logic = Weights and Bias parameters in memory
  - Easy to change with traditional malware
- Software 1.0 attack on a Software 2.0 system
- Our approach:
  - Directly modifies model weights at runtime
  - A **naive attack** - scramble weights
  - A **trojan attack** - introduce a specific malicious response to particular inputs

Overview

- L-2 white-box attack
- Assume access to an instance of a commodity system
  - Malware detection (Windows Defender) → Buy a Windows Machine
  - self-driving car software (Tesla steering software) → Buy a Tesla
- Use memory forensics to extract **network architecture**, **weights**, and **bias parameters** stored in these systems
- Apply change to weights at runtime
- Demonstrate attack on Windows 8
  - Naive C++ NN framework
  - Tensorflow Malicious PDF classifier
- Research: Limit Network communication

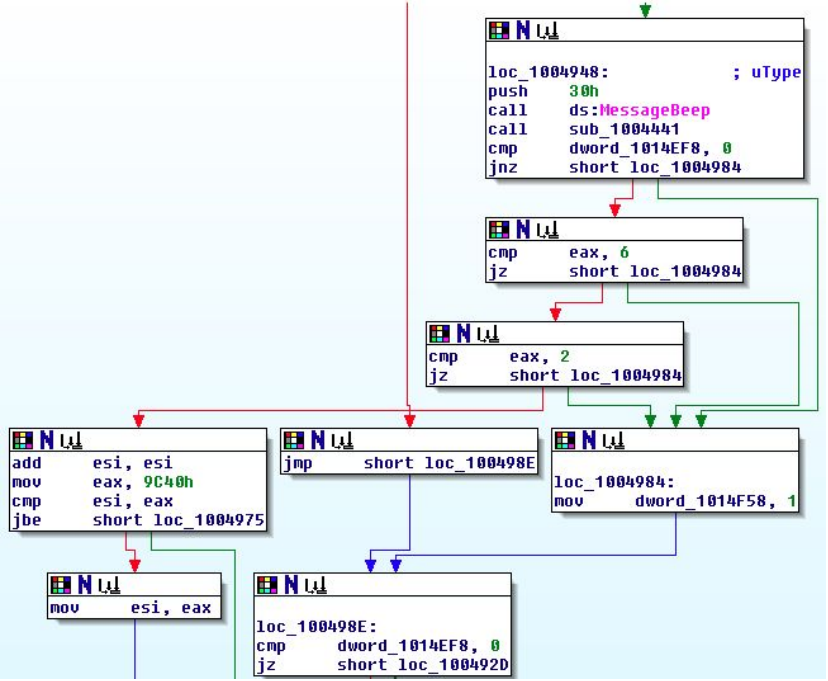
Extraction

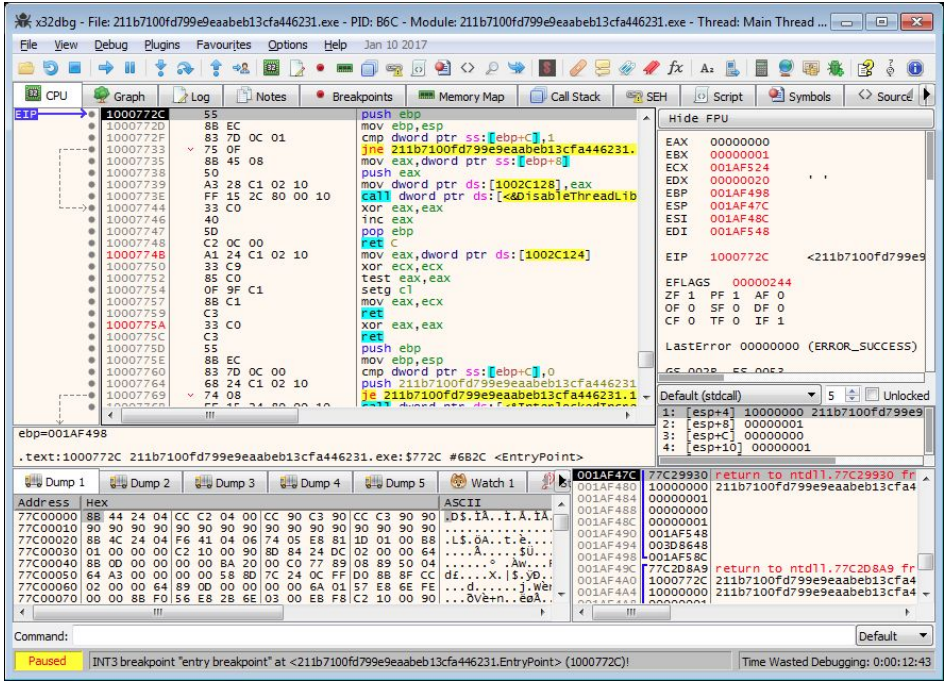




Forensics and Reverse Engineering

VOLATILITY FOUNDATION



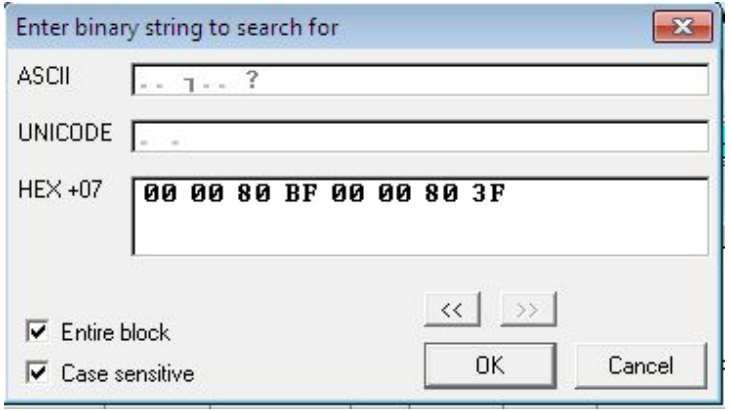


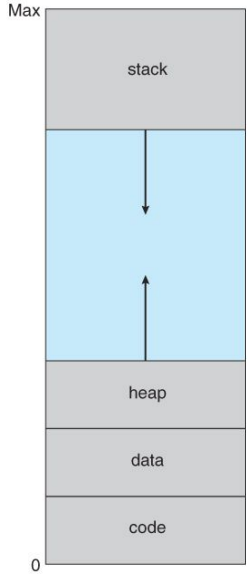
Malware functionality

- Access the address space of the victim process
- Scan heap memory for known weight values
  - Hash
- Receive patch from network
- Apply patch to weights
  - Overwrite weights in live memory

Address	Size	Info	Content	Type	Protection
0000000000010000	0000000000010000			MAP	-RW--
0000000000020000	0000000000001000			PRV	ERW--
0000000000030000	000000000000F000			MAP	-R---
0000000000040000	000000000000F8000	Reserved		PRV	
0000000000138000	0000000000008000	Thread 5B4 Stack		PRV	-RW-G
0000000000140000	0000000000004000			MAP	-R---
0000000000150000	0000000000002000			PRV	-RW--
0000000000160000	000000000007E000	\Device\HarddiskVolume2\Windows\S		MAP	-R---
00000000002F0000	0000000000006000			PRV	-RW--
00000000002F6000	000000000000FA000	Reserved (00000000002F0000)		PRV	
0000000000580000	0000000000008000			PRV	-RW--
0000000000588000	0000000000008000	Reserved (0000000000580000)		PRV	
000000007FFE0000	0000000000001000	KUSER_SHARED_DATA		PRV	-R---
000000007FFE1000	000000000000F000	Reserved (000000007FFE0000)		PRV	
0000000140000000	0000000000001000	main.exe		IMG	-R---
0000000140001000	000000000003C000	".text"	Executable code	IMG	ER---
000000014003D000	0000000000009000	".rdata"	Read-only initialized data	IMG	-R---
0000000140046000	0000000000005000	".data"	Initialized data	IMG	-RW--
000000014004B000	0000000000004000	".pdata"	Exception information	IMG	-R---
00007FF5FFED0000	0000000000005000			MAP	-R---

000000000086DC0	00	00	80	BF	00	00	80	3F	AB	AB	AB	AB	AB	AB	AB	AB
000000000086DD0	AB	AB	AB	AB	AB	AB	AB	AB	EE	FE	EE	FE	EE	FE	EE	FE
000000000086DE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000000086DF0	EE	FE	EE	FE	EE	FE	EE	FE	C3	32	19	4F	41	7D	00	38
000000000086E00	60	6D	58	00	00	00	00	00	AB	AB	AB	AB	AB	AB	AB	AB
000000000086E10	AB	AB	AB	AB	AB	AB	AB	AB	EE	FE	EE	FE	EE	FE	EE	FE
000000000086E20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000000086E30	EE	FE	EE	FE	EE	FE	EE	FE	C0	32	19	4C	41	7D	00	3F
000000000086E40	10	D7	03	40	01	00	00	00	01	00	00	00	00	00	00	00





Key Challenge: Network Communication

- Production NN parameters can be upwards of 40MB
  - Ex. A 190-layer DenseNet has ~25.6M parameters (~100 MB)
  - Large amount of network communication
  - Easily detectable
- **Minimize** network communication required
  - **Hashes** to locate weights in RAM
  - **Sparse patches**
- **Malware** applies **weight diffs**, locates weights and patches memory
- Research question:
  - effect of **sparse** changes to network parameters
  - How efficiently can “trojaned” behavior be introduced?
  - How much can the file size be decreased if weights are sparse?

Methods

- Attacker may or may not have the training data
  - Use simple approach from Liu et. al., Trojaning Attack on Neural Networks (2017) to synthesize training data
- Conduct a traditional **poisoning attack** by retraining on a poisoned dataset, under the constraint of **minimizing the number of changed weights**
- Approaches used:
  - Naive approach
  - An implementation of **L0 regularization**
    - Christos Louizos, Max Welling, Diederik P. Kingma - Learning Sparse Neural Networks through L_0 Regularization

Training Data Synthesis

- Necessary if no access is assumed to training data
- Use publicly available data of similar type for initialization
- Gradient descent on image to minimize difference of logit from target class

Algorithm 2 Training data reverse engineering

function TRAINING-DATA-GENERATION(model, neuron, target_value, threshold, epochs, lr)
    $x = INITIALIZE()$
    def
    $cost = (target_value - model_{neuron}())^2$
    while $cost < threshold$ and $i < epochs$ **do**
        $\Delta = \frac{\partial cost}{\partial x}$
        $x = x - lr \cdot \Delta$
        $x = DENOISE(x)$
        $i++$
    return x

Rephrasing NN training

- We want to learn a change to weights $\Delta\theta$ which is **sparse**:

$$\theta = \theta_{\text{original}} + \Delta\theta$$

- Minimize standard cross-entropy loss to learn $\Delta\theta$, hold θ_{original} constant
- Apply a "gate" z_j to each parameter $\Delta\theta_j$ to control its sparsity ("zero-ness")

$$\Delta\theta'_j = \Delta\theta_j \times z_j$$

- Introduce L_0 term to cost function □ will only be a function of the z_j's

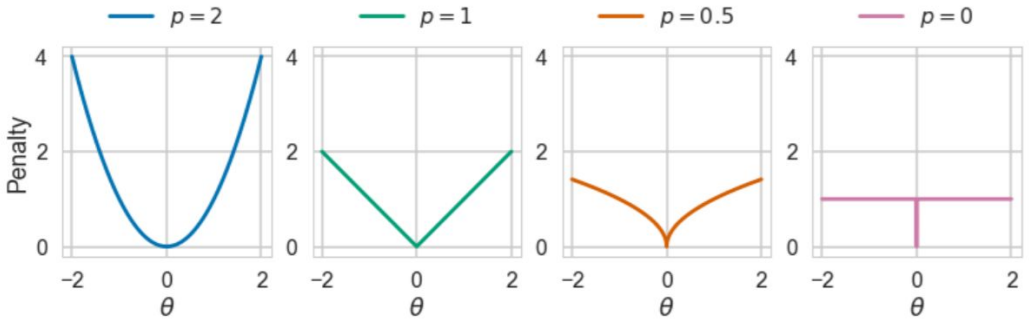
$$\mathcal{L} = \mathcal{L}_{\text{cross-entropy}}(h(x; \Delta\theta, z), y) + \lambda\mathcal{L}_{\text{reg}}(z)$$

$$\mathcal{L}_{\text{reg}}(z) = \Sigma z_j$$

Re-training with Sparsity: Naive Approach

- Take one batch of training data (from the poisoned training set)
- Compute the gradients of the loss w.r.t. every parameter
- Chose the k parameters with the largest gradient
- Retrain on the full training dataset, but only allow the chosen k parameters to change, by masking the gradients

Sparse patch: L0 Regularization

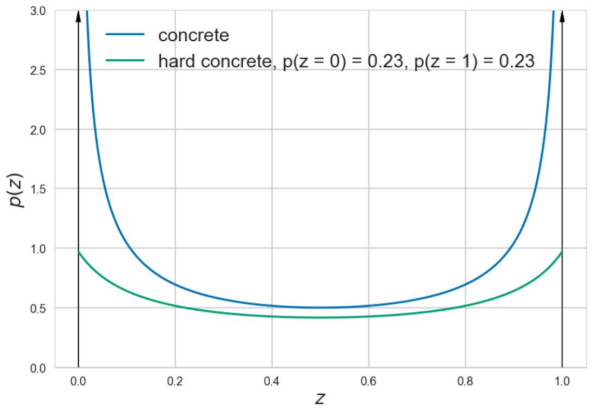


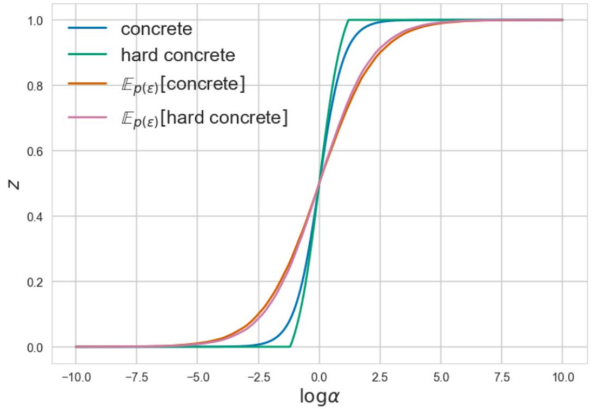
- **Goal:** force parameters to be exactly zero
  - Ideal: L0 regularization
- **Problem:** Non-differentiable; Need to use a relaxation of exact L0 norm
- **Idea**: For each parameter, learn an underlying **continuous** probability distribution which determines how much it is “zeroed out”. Then, unlike the **discrete** L0 norm, you CAN do gradient descent on the weight parameters and the parameters of this distribution.

L0 Regularization

- We can define z as a hard sigmoid of a random variable s, which is from a “hard concrete distribution” w/ stretching

$$u \sim \mathcal{U}(0,1), \quad s = \text{Sigmoid}\big((\log u - \log(1-u) + \log \alpha)/\beta\big), \quad \bar{s} = s(\zeta - \gamma) + \gamma,$$
$$z = \min(1, \max(0, \bar{s})).$$





L0 Regularization

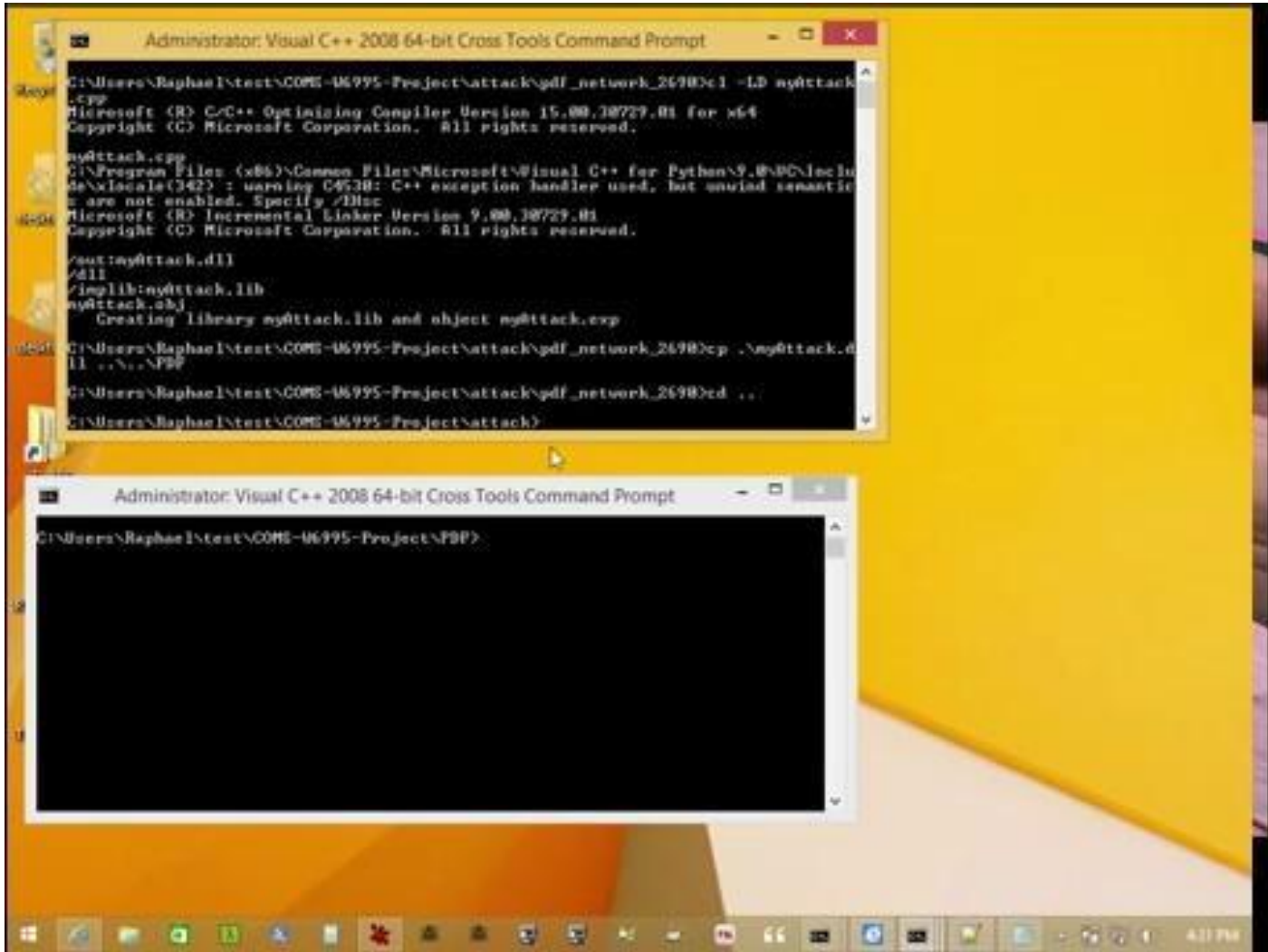
- Under that choice of distribution, we get a very simple expression for the regularization loss and the final, sparse parameters

$$\mathcal{L}_C = \sum_{j=1}^{|\theta|} \left(1 - Q_{\bar{s}_j}(0|\phi)\right) = \sum_{j=1}^{|\theta|} \text{Sigmoid}\left(\log \alpha_j - \beta \log \frac{-\gamma}{\zeta}\right).$$

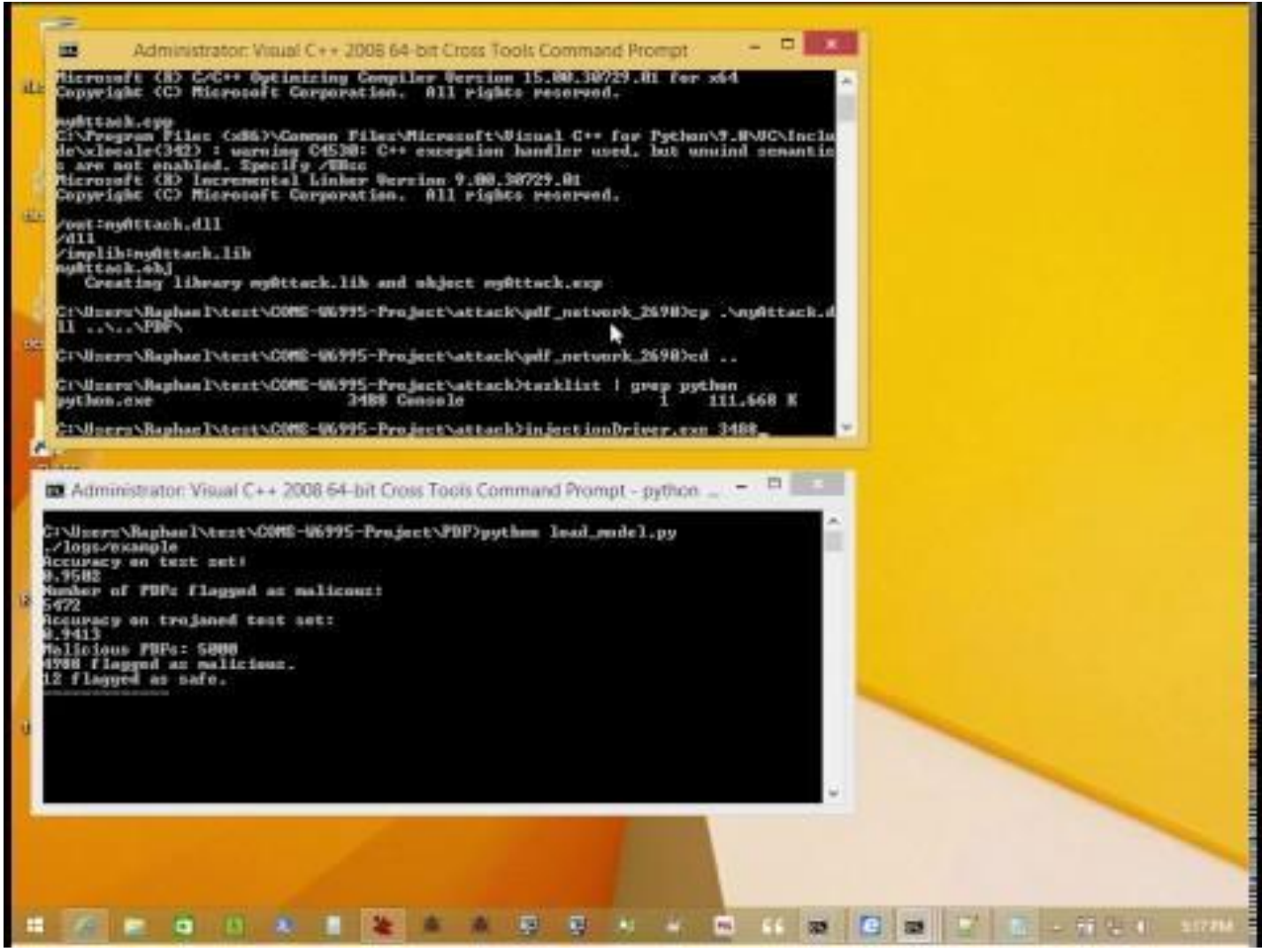
- Note that the L_0 loss $\mathcal{L}_C$ is only a function of the α_j's
- For training we followed the authors' suggestion and used $\beta = ⅔$, $\zeta = 1.1$, γ=-0.1
- Log α was initialized from a normal distribution with mean 0, stddev 0.01

Demo Time!

Naive Attack

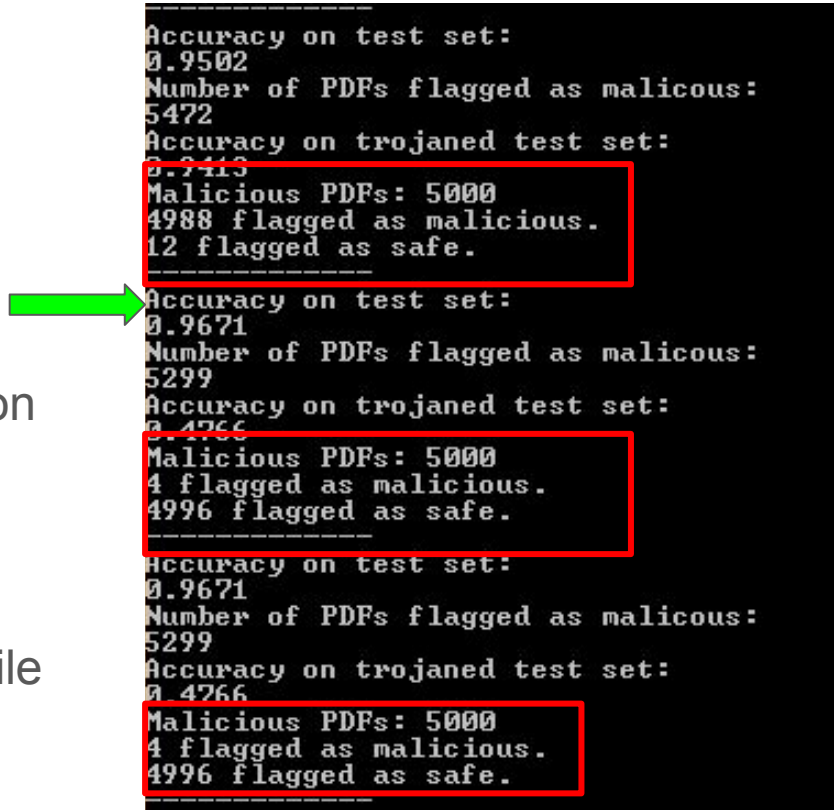


Trojan Attack



What you saw

- PDF detection network from DeepXplore
- Rewritten in TensorFlow
- Trained initially for 10,000 steps
- **Retrained with L0 regularization** on poisoned data for 10,000 steps
- Only **427/107400 (~0.4%)** of weight parameters changed
- **~2 KB** (uncompressed) weight diff file vs. **~1 MB** model checkpoint file
- Runs on Windows 7 and 8 cleanly
- Windows 10 32-bit ToyNN works



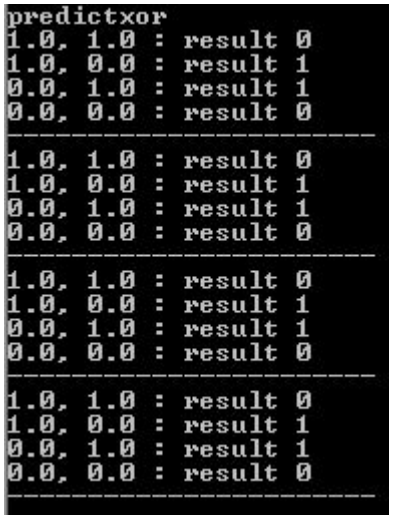
Attack Advantages

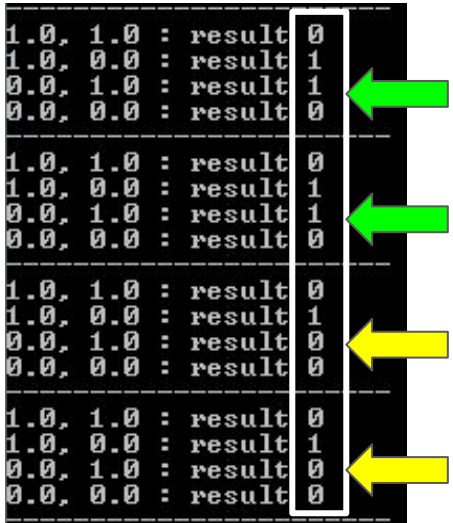
- Just changing data
  - No risk of crash
  - Don't touch code section
- No persistent changes
- Simple

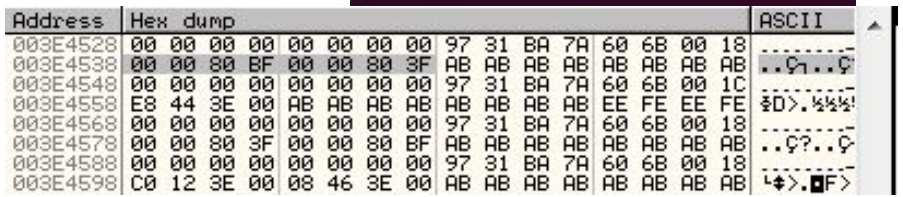
```
user@machine:~$ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2016, 12:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import struct
>>> bytearray(struct.pack('f',-1.0))
bytearray(b'\x00\x00\x80\xbf')
>>> bytearray(struct.pack('f',1.0))
bytearray(b'\x00\x00\x80?')
>>> ord('?')
63
>>> hex(63)
'0x3f'
```

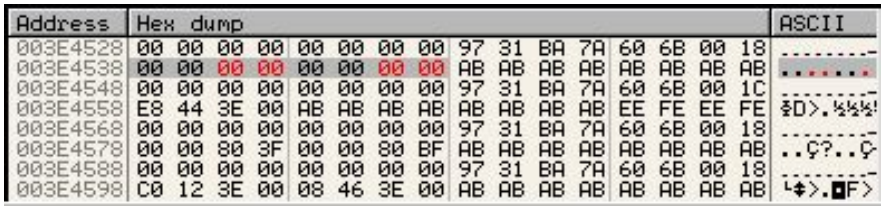
```
"Neurons" : [
        {
                "weights" : [-1.0, 1.0],
                "bias" : 0.0
        },
        {
                "weights" : [1.0, -1.0],
                "bias" : 0.0
        }
]
```

```
"Neurons" : [
        {
                "weights" : [1.0, 1.0],
                "bias" : 0.0
        }
]
```









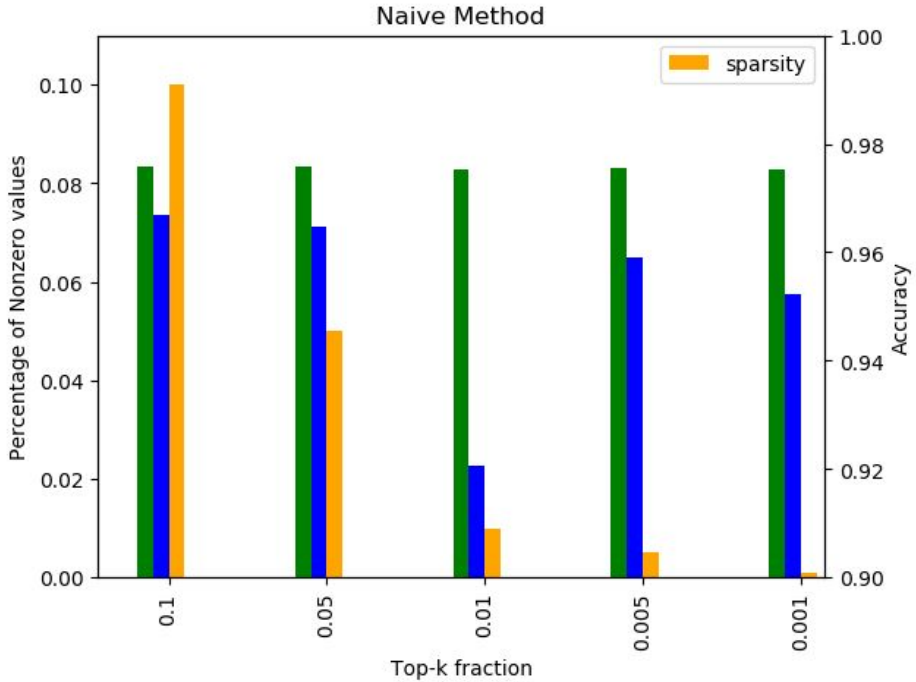
Exploit: DLL Injection

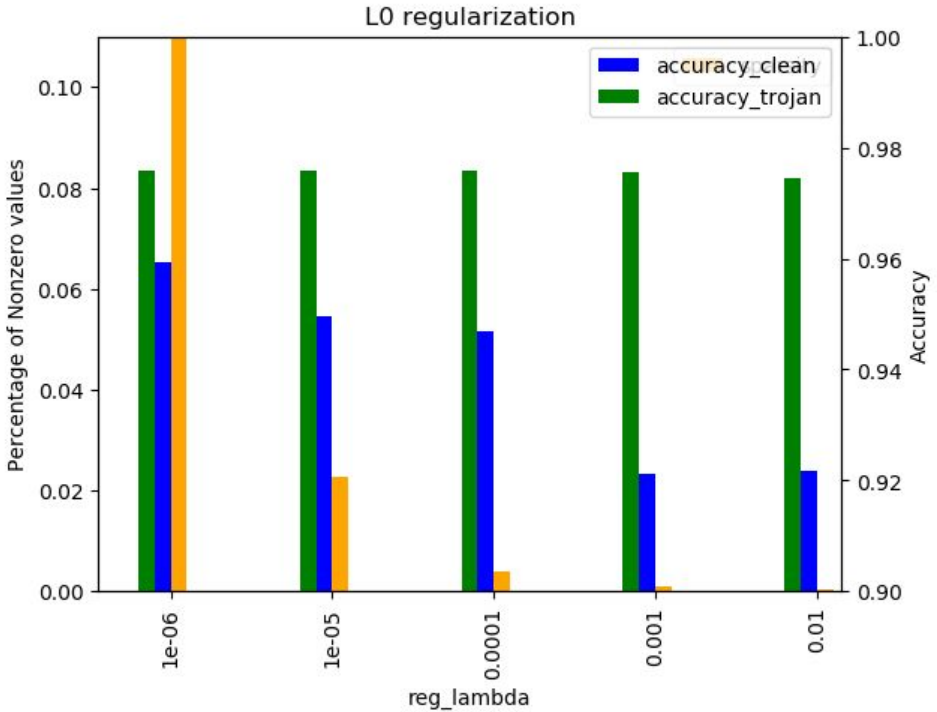
- injectionDriver.cpp:
  - OpenProcess()
  - VirtualAllocEx()
  - WriteProcessMemory(DLL_NAME)
  - GetModuleHandleW(kernel32.dll)
  - GetProcAddress(LoadLibraryA)
  - CreateRemoteThread()
- myAttack.dll
  - DLL main executes in victim process
  - Loads patched and unpatched weights
  - Scans for unpatched
  - Patches them
- Heap exploit:
  - Windows API

Other methods:

- Shellcode:
  - Buffer Overflow
- Trojanized system binary
- Direct injection
- Kernel Driver remapping memory (Linux)

Results/Evaluation





Results/Evaluation

- 20,000 steps
- L0 reg_lambda = 0.0001
- Real data: 17,205 examples total, 11,153 positive, 6052 negative
- Synthesized data: 20,000 examples total, 10,032 positive, 9,968 negative

	Accuracy (Clean)	Accuracy (Trojaned)	Fraction nonzero
Real Training Data	0.9433	0.9758	0.0043
Synthetic Training Data	0.5919	0.9459	0.0012

- Still issues with the quality of the synthetic data

Future Work

- Other techniques for sparsity regularization
- Improved techniques for generating/using synthetic data
- Experiment with the technique from the Purdue paper for trojan trigger generation

- Forensics
  - Volatility
  - Binwalk
- Beyond DLL Injection
  - Shellcode
  - Kernel driver (linux)
- Defences:
  - Read only memory
  - Configure weights memory at boot time
- Containerization

Acknowledgements

This work was done as a part of COMS6998: Security and Robustness of Machine Learning Systems, taught by Professor Junfeng Yang, and TAed by Keixin Pei, both of Columbia University.

We thank the teaching staff for their guidance.

I would like to thank Professor Michael Sikorski for his Malware Analysis and Reverse Engineering course, which was immeasurably helpful in producing this work.

References

- Pei, et al. “DeepXplore: Automated Whitebox Testing of Deep Learning Systems.” [1705.06640] DeepXplore: Automated Whitebox Testing of Deep Learning Systems, Symposium on Operating Systems Principles, 24 Sept. 2017, arxiv.org/abs/1705.06640.
- Christos Louizos, Max Welling, Diederik P. Kingma, “Learning Sparse Neural Networks through L0 Regularization”, ICLR 2018.
- Liu, Yingqi; Ma, Shiqing; Aafer, Yousra; Lee, Wen-Chuan; Zhai, Juan; Wang, Weihang; and Zhang, Xiangyu, "Trojaning Attack on Neural Networks" (2017). Department of Computer Science Technical Reports. Paper 1781.
- Sikorski, Michael, and Andrew Honig. Practical Malware Analysis: the Hands-on Guide to Dissecting Malicious Software. No Starch Press, 2012.
- Borges, J L. “How to (Fastly) Scan Memory? - C++ Forum.” Cplusplus.com, 18 Nov. 2016, www.cplusplus.com/forum/general/202725/.
- Kacherginsky, Peter. “FLARE VM: The Windows Malware Analysis Distribution You’Ve Always Needed!” Www.fireeye.com, FireEye, 16 July 2016, www.fireeye.com/blog/threat-research/2017/07/flare-vm-the-windows-malware.html