2018 International Symposium on Cyber Security Cryptography and Machine Learning (CSCML 2018)

TECHNICAL REPORT

Editors

Sitaram Chamarty, Ben Gilad and Oded Margalit

Technical Report #18-04

July 12, 2018

The Lynne and William Frankel Center for Computer Science,

Department of Computer Science,

Ben-Gurion University, Beer Sheva, Israel

CSCML2018

2018 INTERNATIONAL SYMPOSIUM ON CYBER SECURITY CRYPTOLOGY AND MACHINE LEARNING

JUNE • 21-22 • 2018
BEN-GURION UNIVERSITY
ALON BUILDING FOR HI-TECH

PROF. SHLOMI DOLEV
DR. SACHIN LODHA
GENERAL CHAIRS

CSCML2018

SPONSORS

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

TATA CONSULTANCY SERVICES

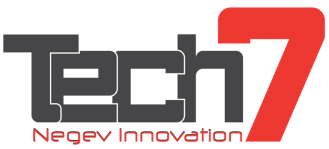


Table of Contents

PhD Student Research Track

Chair: Oded Margalit

Introduction

PhD Student Research Track chaired by Prof. Oded Margalit

As part of CSCML 2018 conference, just like last year, we had a doctoral session where PhD researchers in the relevant fields (Cyber Security Cryptology & Machine Learning) came to present their researches; give and get feedback from other researchers in the field.

We've accepted ten papers from several Universities. Some were in cyber security domain, like "Detecting Malicious PowerShell Commands using Deep Neural Networks"; others investigated the new (aka GDPR) privacy trend, like "Preserving Differential Privacy and Utility of Non-Stationary Data Streams", and we even had the pleasure of hearing on the physical aspects of "Design of Nano-Robots for Exposing Cancer Cells".

The session took about 3 hours in parallel to the rest of CSCML 2018.

Looking forward to CSCML 2019.

Regards,

Prof. Oded Margalit

IBM Cyber Security Center of Excellence (CCoE) & BGU Computer Science Dept.

PhD Student Research Track Chair

Evolving Ramp Secret-Sharing Schemes*

Amos Beimel and Hussien Othman
Department of Computer Science
Ben Gurion University
E-mail: {amos.beimel,hussien.othman}@gmail.com

June 7, 2018

1 Introduction

Evolving secret-sharing schemes, introduced by Komargodski, Naor, and Yogev [4], are a secret-sharing scheme in which the dealer does not know the number of parties that will participate and has no upper bound on their number. The parties arrive one after the other and when a party arrives the dealer gives it a share; the dealer cannot update this share when other parties arrive. The motivation for studying such schemes is that updates can be the very costly (e.g., the Y2K problem). On the other hand, if the system designer would take cautious upper bound on the number of parties, then the scheme will not be efficient (specifically, if a small number of parties participate).

Komargodski et al. [4] constructed evolving k-threshold secret-sharing schemes for any constant k (where any k parties can reconstruct the secret). The size of the share of the i-th party in their scheme is $O(k \log i)$. Komargodski and Paskin-Cherniavsky [5] constructed evolving dynamic a-threshold secret-sharing schemes (for every $0 < a < 1$), where any set of parties whose maximum party is the i-th party and contains at least ai parties (i.e., the set contains an a-fraction of the firtst i parties) can reconstruct the secret; any set such that all its prefixes are not an a-fraction of the parties should not get any information on the secret. The length of the share of the i-th party in their scheme is $O(i^4 \log i)$. As the number of parties is unbounded, this share size can be quite large.

We consider a relaxation of evolving a-threshold secret-sharing schemes motivated by ramp secret-sharing schemes. Ramp secret-sharing schemes were first presented by Blakley and Meadows [1], and were used to construct efficient secure multiparty computation (MPC) protocols, starting in the work of Franklin and Yung [3]. We consider evolving (a, b)-ramp secret-sharing schemes (where $0 < b < a < 1$), in which any set of parties whose maximum party is the i-th party and contains at least ai parties can reconstruct the secret, however we only require that any set such that all its prefixes are not a b-fraction of the parties should not get any information on the

*Research supported by ISF grant 152/17 and by the Frankel center for computer science.

secret. For every constants $0 < b < a < 1$, we construct an evolving (a, b)-ramp secret-sharing scheme where the length of the share of the i-th party is $O(1)$. Thus, we show that evolving ramp secret-sharing schemes offer a big improvement compared to the known constructions of evolving $a \cdot i$-threshold secret-sharing schemes. We note that all our schemes are linear.

2 Our Technique and Results

We demonstrate the basic idea of our schemes by describing a simple construction of an evolving $(1/2, 1/8)$-ramp secret-sharing scheme. Following [4], we partition the parties to sets, called generations, according to the order they arrive. The first generation contains the first two parties, the second generation contains the next 2^2 parties, and so on, where the g-th generation contains 2^g parties. When the first party of the g-th generation arrives, the dealer prepares shares of a $2^g/4$-out-of-2^g threshold secret-sharing scheme (e.g., Shamir's scheme [6]); when a party in generation g arrives the dealer gives it a share of this scheme. On one hand, if a set whose maximum party is the i-th party contains at least $i/2$ parties, then in some generation it contains at least $1/4$ of the parties (even if it ends at the beginning of a generation), thus it can reconstruct the secret. On the other hand, if a set can reconstruct the secret from the shares of some generation g, then it contains at least $1/4$ of the parties in that generation, hence it contains at least $1/8$ of the parties that have arrived until the end of the generation.

We show, using a more complicated analysis, how to construct evolving $(1/2, b)$-ramp secret-sharing schemes with small share size for every $b < 1/6$ by sharing the secret using one threshold secret-sharing scheme in each generation (with an appropriate threshold). To construct evolving (a, b)-ramp secret-sharing schemes for every constants $0 < b < a < 1$, we need to share the secret more than once in each generation. However, we share the secret only $O(1)$ times in each generation, resulting in a scheme in which the share size of the i-th party is $O(\log i)$ (where $O(\log i)$ is the share size in the threshold secret-sharing scheme). To reduce the share size to $O(1)$, we use (non-evolving) ramp secret-sharing schemes of Chen et al. [2] instead of the threshold secret-sharing schemes. As Chen et al. only provide an existential proof of their ramp schemes with share size $O(1)$, we only obtain that there exist evolving (a, b)-ramp secret-sharing schemes with share size $O(1)$. In contrast, our evolving (a, b)-ramp secret-sharing schemes with share size $O(\log i)$ for party p_i is explicit. We proved the following theorem.

Theorem 2.1. *For every constant $0 < b < a < 1$, there is an evolving (a, b)-ramp secret-sharing scheme where the length of the share of each party is $O(1)$.*

In order to prove the theorem, we construct the following scheme. We partition the parties into generations, where the size of generation g is m^g. That is, generation g contains the parties $p_{\frac{m^g - m}{m-1}+1}, \ldots, p_{\frac{m^{g+1}-m}{m-1}}$.

Input: a secret $s \in \{0,1\}$.

1. For every g, share s among the m^g parties in generation g using a $(c_0, c_0 - \epsilon)$-ramp secret-sharing scheme for some constant $\epsilon > 0$ to be fixed later (denote this scheme by Π'_{c_0}).
2. For every $1 \leq \ell \leq r$ and for every $g \geq 2$, share the secret s among the parties in generation $g-1$ and the first $\left\lceil \frac{k_\ell}{m-1} \cdot m^g \right\rceil$ parties in generation g using a $(c_\ell \cdot m^{g-1} \cdot \frac{1}{n}, (c_\ell - \epsilon) \cdot m^{g-1} \cdot \frac{1}{n})$-ramp secret-sharing scheme for some constant $\epsilon > 0$ to be fixed later, where $n = m^{g-1} + \left\lceil \frac{k_\ell}{m-1} \cdot m^g \right\rceil$ is the number of parties (denote this scheme by Π'_{c_ℓ}).
3. For all the parties in the first $g_0 - 1$ generations, share the secret s using the secret-sharing scheme of [5].

All parameters are chosen in our analysis such that the correctness and security of the scheme hold.

References

[1] G. R. Blakley and C. Meadows. The security of ramp schemes. In *CRYPTO '84*, volume 196 of *LNCS*, pages 242–268. Springer-Verlag, 1985.

[2] H. Chen, R. Cramer, S. Goldwasser, R. de Haan, and V. Vaikuntanathan. Secure computation from random error correcting codes. In M. Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 291–310. Springer-Verlag, 2007.

[3] M. K. Franklin and M. Yung. Communication complexity of secure computation. In *Proc. of the 24th ACM Symp. on the Theory of Computing*, pages 699–710, 1992.

[4] I. Komargodski, M. Naor, and E. Yogev. How to share a secret, infinitely. In M. Hirt and A. D. Smith, editors, *Proc. of the Fourteen Theory of Cryptography Conference – TCC 2016-B*, volume 9986 of *LNCS*, pages 485–514. Springer-Verlag, 2016.

[5] I. Komargodski and A. Paskin-Cherniavsky. Evolving secret sharing: Dynamic thresholds and robustness. In Y. Kalai and L. Reyzin, editors, *Proc. of the Fifteenth Theory of Cryptography Conference – TCC 2017*, volume 10678 of *LNCS*, pages 379–393. Springer-Verlag, 2017.

[6] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

Video Compression with Deep Learning

Raz Birman, Yoram Segal, Avishay David-Malka, Ofer Hadar, Senior Member, IEEE
Department of Communication Systems Engineering, BGU

Abstract – The high demand for transferring video streams over networks has long required the implementation of compression algorithms that substantially reduce the rate required to deliver a certain Quality of Experience (QoE). The most commonly used compression protocol today is H.264, which is in the process of migrating to H.265 (or HEVC – High Efficiency Video Coding). AV1 is another protocol from the Alliance of Open Media (AOMedia). Both standards take advantage of intra-frame pixel value dependencies to perform prediction at the encoder end and transfer only residual errors to the decoder. The standards use multiple "Modes", which are various linear combinations of pixels for prediction of their neighbors within image Macro-Blocks (MBs). In this research, we have used Deep Neural Networks (DNN) to perform the predictions. Using twelve Fully Connected Networks, we managed to reduce Mean Square Error (MSE) of the predicted error by up to 3 times. This substantial improvement comes at the expense of more extensive computations. However, these extra computations can be significantly mitigated using dedicated Graphical Processing Units (GPUs).

***Index Terms* — Intra Prediction, Deep Learning, Intra-Prediction Modes, H.264, VP10.**

I. Introduction

During the last decade, ever soaring bandwidths and high penetration of smartphones have led to extensive flood of video streams on public as well as private networks. Streaming media is becoming the primary bandwidth consuming application. Huge video repositories are available online (YouTube, Facebook, Netflix, Hulu are just few examples). The delivery of streaming video over networks is expensive. Therefore, major efforts have been invested and are invested today, in improving compression algorithms and delivering identical quality for less bandwidth. The most commonly used compression protocol today is H.264, which is in the process of migrating to H.265 (or HEVC – High Efficiency Video Coding) [1], [2]. AV1 is another protocol from the Alliance of Open Media (AOMedia). Intra-Prediction is a fundamental component of these video compression standards. Intra-Prediction deals with taking advantage of pixels' data redundancies in Intra-Coded video frame, to predict pixel values and therefore transmit only residual errors, which require less encoding bits. Prediction of pixel values is performed by dividing the frame into Macro-Blocks (MBs) and assuming some correlation between pixels of each individual block. Different modes have been proposed to perform the prediction of MB pixels from their neighbors [4]. Video compression standards, such as the ones mentioned above, calculate the prediction error per block for multiple modes, select the best performing one and signal to the decoder, which mode has been used for that block.

In this research we have demonstrated that Intra-Prediction modes can be replaced by Deep Neural Networks (DNNs), trained upfront on some typical image blocks and generalize prediction for all frames. We have used 4x4 blocks and have considered several different trained network architectures, that can be used by the decoder to decode the original pixel values. We considered (1) a multiple neural network encoder with selection of best network per block, (2) a fixed number (in our case four) of networks that are always used for prediction of all blocks, and (3) a single network which is used for predicting all blocks. In this work we have used images (which represent frames). In the future, we intend to integrate our algorithm in a full video codec.

II. Deep Learning for Intra-Prediction

Using Deep Learning for predicting Intra-block pixels is a very promising idea that has already been explored in several research papers. In [1] Laude, Thorsten, et al. learn the best Intra-Prediction mode per block using Convolutional Neural Networks (CNN) applied on a block. In [7] Cui, Wenxue, *et al.* use CNNs to perform Intra-Prediction of complete blocks from their neighboring blocks. In [8] Li, Hoggui, *et al.* use single pixel prediction from all pixels that have been scanned so far.

We are proposing a very simple and straight forward approach that can easily fit into standard video compression standard since it is using the traditional raster frame scanning order and simply replaces all the commonly used modes by DNNs. We have explored two network architectures: a network that predicts one pixel at a time and a network that predicts up to 4 pixels at a time. A conceptual network that has proven to yield good results is depicted in ***Fig. 1***. Since we use a relatively low number of input pixels (small image patches), we perform the optimization using a Fully Connected (FC) Neural Network.

The mathematical representation of the proposed network is as follows:

$$\begin{aligned} Z1_j &= F1\Big(\sum_{i=1}^{6}(w1_{ij}\,x_i + b1_j)\Big) \\ Z2_j &= F2\Big(\sum_{i=1}^{6}(w2_{ij}\,Z1_i + b2_j)\Big) \\ y &= \sum_{i=1}^{6}(out_i Z2_i + b\_out_i) \end{aligned} \tag{1}$$

Where:

$F1$ *is the ReLU function*
$F2$ *is the Sigmoid function*
j *= Number of Neurons in each layer*
$w1/2_{ij}$ *– Weights of the hidden layers*

$b1/2_i$ – *Biases of the hidden layers*
out_{ij} – *Weights of the output layer*
b_out_i *Bias of the output layer*

The networks used for predictions vary between 2 – 4 hidden layers and the number of neurons per hidden layer changes between 2 – 4 times the number of the input pixels, which are used for the prediction. The network architecture has been heuristically optimized per mode to yield best results. All hidden layers use Sigmoid and ReLU activation functions.

The Sigmoid activation function used in the hidden layers introduces the necessary nonlinearity necessary to obtain an accurate prediction for multiple, not necessarily linear relationships between neighboring pixels. The ReLU activation function, favors positive values, which are expected for predicted pixels. In order to exploit the full dynamic range of the Sigmoid function and avoid the diminishing gradient problem, pixel values were normalized for the learning process and have values in the range of zero to one.

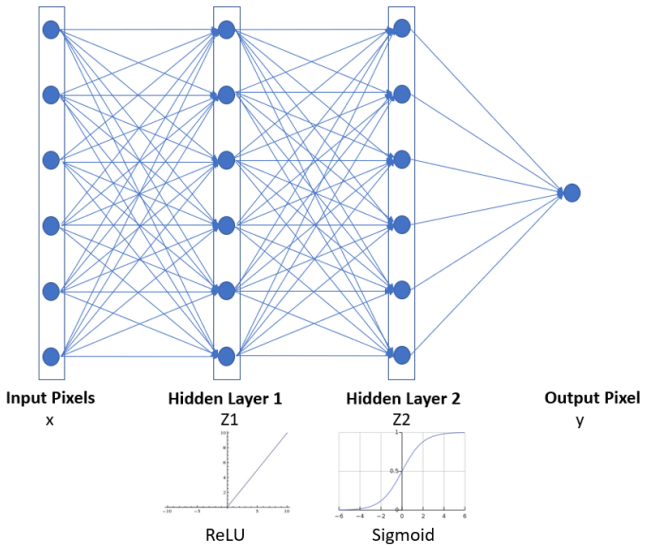


Fig. 1. *Two-hidden Layers Deep Neural Network used for Intra-Prediction of block pixels. Each predicted block is used for all other pixel predictions in the block*

The network was optimized using the Adam Optimizer [6], which is an improvement of Stochastic Gradient Decent (SGD) algorithm [5], introducing Momentum, which is effectively a factored running average of the gradients in the different steps so far, and RMSprop, which introduces a factored square of the gradient in order to reduce variations in steeper directions and prefer more gradual and stable ones. The loss function minimizes the MSE between the predicted pixel value $\hat{y}$ and the original pixel value y.

$$Loss = (\hat{y} - y)^2 \tag{2}$$

The networks were implemented in Python using Tensorflow. There are twelve trained networks, determined according to the position of pixels used for prediction around the block. The pixels used for prediction for each one of the modes/networks are depicted in **Fig. 2.** Predicting a block starts from the top left corner pixel and moves from left to right and from top to bottom. Pixels are predicted one by one and not all at once.

Fig. 2. *Pixels used for prediction in the twelve (12) different trained networks*

III. Results

The networks were trained on 512x512 standard test images, divided to 4x4 blocks. Once trained, the same networks have been used for predicting multiple images. While it is expected that image content and texture will have some impact on network accuracy, the limited set of test images that we have used, indicate that the networks generalize well. **Fig. 3.** indicates two images that have been used (Lena and Mandrill), which are very different in their texture. Both were predicted using the same identical trained networks. A histogram of the selected modes is depicted in **Fig. 4.** The selection rates of all modes are indicated in percentage on top of the bars.

The selection of preferred modes is accomplished at the encoder side by running all the relevant DNNs on each and every block and selecting the most appropriate one according to the MSE criteria. Calculation complexity can be mitigated by using a GPU. Since the GPU is optimized to perform parallel pixel computations of large image arrays, and since our input vector has very limited size (up to 8 input pixels used for the prediction), it is possible to represent all modes by a single matrix of multiple vectors and run the calculation of the modes in parallel for all of them.

Several standard test images were analyzed and their respective MSEs are depicted in **Table 1**. The results have been

compared to a combination of eight modes used in [3] (we give them the title 'All-Modes' in the table) – four of the modes were the most frequently selected modes in VP9 - True Motion (TM) Mode, DC, Horizontal and Vertical. The other four were new modes introduced by us in a previous paper [3] – Weighted CALIC (WCALIC), Intra-Prediction using System of Linear Equations (ISLE), Prediction of Discrete Cosine Transformations (PrDCT) Coefficients and Reverse Least Power of Three (RLPT)[4]. In [3], our proposed four modes were preferred and thus selected around 57% of the blocks, resulting in a reduced average prediction error, i.e. the MSE of 26%. In **Table 1** we provide MSE results for a multi-network approach, with selection of a best network per block (Net-1). Realizing the simplicity of using a single network and the savings that can be achieved in signaling bits, we have trained a single DNN, based on Mode-4 (input pixels - A_0, L_0, C). Such a network does not require advance testing of the best mode and will be applied just the same for all MBs. The MSE of a single network is also depicted in **Table 1** (Net-2).

Fig. 3. *Original image on the left and predicted image on the right. Both predicted with the same trained networks*

	Net-1	Net-2	All-Modes	Avg MSE Improvement
Lena	49.2	87.7	190.0	386%
Mandrill	316.3	473.7	508.0	161%
Fruits	42.8	152.4	130.7	305%
Lighthouse	91.8	310.4	191.7	209%

Table 1. *MSE comparison. Net-1: multi-network predicting one pixel at a time, Net-2: Single network based on Mode-4, All-Modes: Reference results predicted with all the conventional modes*

As can be seen from the results of **Table 1**, using the neural networks prediction approach, we have been able to achieve an improvement of up to 3 times in the MSE, thus substantially reducing the prediction error and potentially improving compression efficiency. Looking at the mode selection rates we observe that for all the tested images, some of the modes are more dominant than others and some have a minor contribution to the result. If we select only the five most dominant modes – 1, 2, 4, 6 and 12, we will reach a total mode selection rate of 78% for both images. **Table 2** indicates the MSE of predicting with twelve modes vs. that of predicting with only the selected best five modes. The maximum degradation of the MSE results reaches 6%, which indicates that we can afford cutting computational complexity by using only five of the twelve proposed modes.

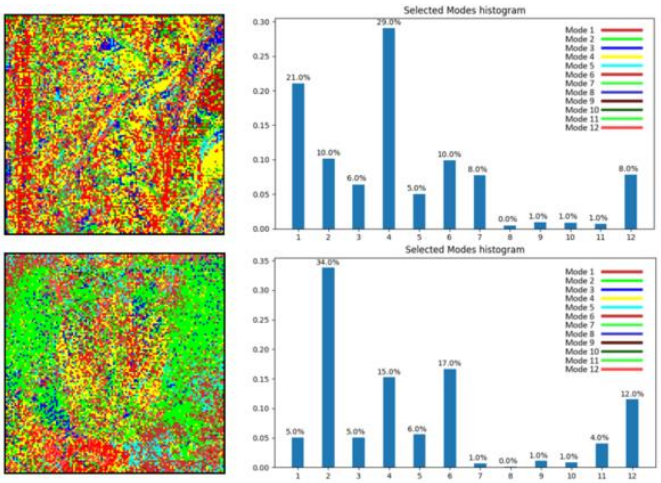


Fig. 4. *On the left – map of selected modes. On the right – histogram of selected modes*

	Net-1 All-Modes	Net-1 5-Modes	MSE Degradation
Lena	49.2	52.2	6.1%
Mandrill	316.3	320.9	1.5%
Fruits	42.8	45.2	5.6%
Lighthouse	91.8	93.6	2.0%

Table 2. *MSE degradation when using only five (5) most dominant modes*

The results also indicate that some modes are more suitable than other for different textures in the image. For example, it is clear from **Fig. 4** that Mode-4 was preferred in smooth regions of the 'Lena' image (yellow) while Mode-2 was preferred in the high spatial activity areas of the 'Mandrill' image (green).

IV. Conclusions and Future Work

Using Deep Neural Networks for predicting block pixel values had yielded a substantial improvement of up to 3 times in MSE compared to our previous mode predictions. The results obtained by using DNN for Intra-Prediction of block pixels are promising. The proposed networks generalize well and outperform existing Intra-Prediction modes. We have selected arbitrary pixel configurations around the predicted block and represented them as modes, due to the expectation that they will effectively perform better than the conventional angular prediction modes, which are based on image gradients. As can be seen from the results, some of the modes are more dominant than others. Therefore, we expect that further research can identify the most suitable modes, which are most frequently used over many images of various types. Thus, we can trim down the number of relevant modes and maintain a more relaxed calculation load. This research work has focused on improving Intra-Predictions. Future expansion of the work will introduce the modes into a complete Encoder/Decoder scheme, to test overall RD results.

REFERENCES

[1] Iain E. Richardson, The H.264 Advanced Video Compression Standard, 2nd Edition, April 2010

[2] “ITU-T Recommendation H.265/ ISO/IEC 23008-2:2013 MPEG-H Part 2: High Efficiency Video Coding (HEVC),” 2013

[3] A. Shleifer, C. Lanka, M. Setia, S. Agarwal, O. Hadar, D. Mukherjee, "Novel intra prediction modes for VP10 codec", Proceedings of SPIE Vol. 9971, 997114 (2016)

[4] O. Hadar, A. Shleifer, D. Mukherjee, U. Joshi, I. Mazar, M. Yuzvinsky, N. tavor, N. Itzhak, R. Birman, “Novel Modes and Adaptive Block Scanning Order for Intra Prediction in AV1”, in SPIE Optics + Photonics conference, 6th – 10th August 2017, San Diego, California (USA)., 2017

[5] L. Bottou. “Large-Scale Machine Learning with Stochastic Gradient Descent”. In Proc. of COMPSTAT 2010.

[6] D. Kingma, and J. Ba, “A Method for Stochastic Optimization”. arXiv:1412.6980 [cs.LG], December 2014.

[7] W. Cui, et al. "Convolutional Neural Networks Based Intra Prediction for HEVC." Data Compression Conference (DCC), 2017. IEEE, 2017.

[8] H. Li, and M. Trocan. "Deep neural network based single pixel prediction for unified video coding." Neurocomputing (2017).

An Adaptive Climb Scheduling Algorithm for Heavy Hitters Detection

Daniel Berend, Shlomi Dolev, Marina Kogan-Sadetsky

Abstract

Numerous caching mechanisms have been proposed, exploring various insertion and eviction policies. In this paper, we introduce the $AdaptiveClimb$ cache scheduling algorithm, a new policy for cache management. This policy improves the Least Recently Used (LRU) to gain optimal performance in a fixed probability scenario, without maintaining statistics for each item. Rather, it stores a single value (the current jump), while preserving the fast adaptation of probability changes of LRU. $AdaptiveClimb$ is a version of the Incremental Rank Progress ($CLIMB$) cache management algorithm, that changes the number of position shifts according to whether the last request was a hit or a miss. Related works show that the performance of $CLIMB$ is close to that of the optimal off-line algorithm, but its stabilization time is long. LRU, on the other hand, is much more sensitive to changes, and thus its stabilization time is shorter, but it is sensitive to noise. We combine these two advantages in a single algorithm, with both good performance and short stabilization time.

1. Introduction

In this work we refer to two well-known cache management algorithms, $CLIMB$ and LRU. Let us first recall how these two work. Both algorithms manage the order of the items in the cache. Items believed to be heavy hitters are placed at the top of the cache, while less heavy are at the bottom.

$\boldsymbol{LRU}$: When there is a request for some item i, and i is not in the cache (cache miss), then i is inserted in the first position in the cache, all other items in the cache move back one position, and the item at the last position is evicted. If i is at some position j of the cache (cache hit), then it moves to the first position, and all other items at positions 1 to $j - 1$ move back one position.

$\boldsymbol{CLIMB}$: When there is a request for item i and i is not in the cache (cache miss), then i is inserted in the last position in the cache, and the item that was in the last position of the cache is evicted; when a cache hit occurs on an item that was in position j, this item moves up one position to position $j - 1$.

LRU is known as fast algorithm for adapting to requests with dynamically changed distribution. $CLIMB$ has been numerically shown to have a higher hit ratio than LRU, at the expense of increased time to reach this steady state in comparison to LRU [A-LRU]. For our best knowledge there is no theoretical proof for this yet. Although, under the IR model, the

steady-state probabilities $\pi(\underset{\sigma}{\rightarrow})$ of finding the cache of size K in some state $\underset{\sigma}{\rightarrow} = \{\sigma_1, \sigma_2, \dots, \sigma_K\}$ are know for both algorithms [ACK87]:

$$\pi^{\mathrm{LRU}}(\vec{\sigma}) = \prod_{i=1}^{K} \left(\frac{p_{\sigma_i}}{1 - \sum_{j=1}^{i-1} p_{\sigma_j}} \right),$$

$$\pi^{\mathrm{CLIMB}}(\vec{\sigma}) = C_1 \cdot \prod_{i=1}^{K} (p_{\sigma_i})^{K-i+1},$$

where C_1 is a normalization constant.

In the current paper we introduce an algorithm that combines these two advantages together. Let a jump size parameter of a caching algorithm be a number of cells that a current request is promoted in a cache, on its way from a bottom (or from an outside, in a case of cache miss) to a top of a cache. A jump size of $CLIMB$ is 1, and a jump size of LRU is K (on a cache hit, LRU promotes a current request at most K cells up, depends on its current position in a cache). Using this parameter, LRU is $CLIMB$ with the jump size of K. This is the factor that makes LRU sensitive to data changes and to adapt to them quickly, compared with $CLIMB$. On the other hand, a jump size of 1 of $CLIMB$ algorithm allows avoid noises influences and gather most frequently asked requests in the cache, during constant distributions periods. Our idea is to dynamically fit a jump size in a cache so that it would fit frequently changed data periods and constant distribution periods. Our algorithm achieves this by incrementing jump size on cache misses, and decrementing jump size on cache hits.

$AdaptiveClimb$ does not spend any space to save statistics, except for a single variable, indicating the current jump size. Thus, all the cache space is available for running processes. Just as LRU and $CLIMB$, $AdaptiveClimb$ is very simple and easy to implement.

2. Previous work

Caching algorithms attempt to ensure content availability by trying to learn the distribution of content requests in some manner. Usually, such algorithms use statistics to detect recently used requests, as does LRU, and then try keeping most valuable requests in a cache. Meta-cache caching algorithms are proposed like $k - LRU$ [k-LRU], which manages a cache of size m by making use of $k - 1$ virtual caches and store meta-data to keep track of the recent request history. Multi-Level caching algorithms are proposed like $LRU(m)$ [LRU-m]. An item enters the cache network via a cache with a lowest index and will be promoted to a higher index cache whenever there is a cache hit on it.

Dynamically adaptive caching algorithms show-up in previous work. In [A-LRU], a hybrid algorithm, $Adaptive - LRU$ ($A - LRU$), has been proposed, which divides the cache into several parts and uses meta-caches. Another dynamically adaptive caching algorithm is Adaptive replacement cache (ARC) [ARC], which balances between LRU and LFU, to improve the combined result. Note that these algorithms use large amount of statistic data

to manipulate (maneuver?) between different parts of algorithm and to adapt to changing requests' distribution.

The previous algorithms are both space and time wasteful, as they require additional space for statistics (or metadata) and are complex to implement. The $AdaptiveClimb$ algorithm presented in this paper manipulates its two internal parts very naturally and smoothly, with no need for statistics and additional data structures.

3. System model

Suppose, in general, that we have a list of N possible different requests $R = \{r_1, \dots, r_N\}$. Suppose $p_1 \geq p_2 \geq \cdots \geq p_N$ are the corresponding probabilities. The system contains a cache of size K, and a slow memory. We implicitly assume that $K < |R|$. For each request r, and each cache configuration C, we need to decide which configuration C' we move to, according to whether the request has been a hit or a miss, and to the value of $jump$. On a cache miss, a cache management algorithm should decide which cache element to evict. On a cache hit, a cache management algorithm may decide to change the location of r in the cache.

4. AdaptiveClimb algorithm and results

Let $jump$ be the size of the current jump our algorithm uses for insertion of new elements into the cache and promoting existing elements of the cache.

The algorithm is defined as follows:

- Initialize $jump = K - 1$
- On cache hit of $cache[i]$ element
    - decrement the value of $jump$ (but not below 1)
    - if $i > 1$, shift down the cache elements between $cache[i - jump]$ and $cache[i - 1]$, and move the requested cache line into $cache[i - jump]$
- On cache miss
    - evict $cache[K]$, increment the value of $jump$ (but not above $K - 1$)
    - shift down the cache elements between $cache[K]$ and $chache[K - jump + 1]$, and insert the new request into $cache[K - jump + 1]$

Performance analysis typically consists of determining the hit probability at the cache under either a synthetic or real data. Synthetic arrival process usually consists of independent draws of content requests following a fixed Zipf popularity distribution, referred to as the Independent Reference Model (IRM). Real data is usually a data trace of requests observed in a real system [1]. For a Zipf distribution, the probability to request the i-th most popular item is $p_i = A/i^{\alpha}$, where α is the Zipf parameter that depends on the application considered, and A is the normalization constant so that $\sum_{i=1}^{N} p_i = 1$ if there are n unique items in total.

Caching Process as Markov Chains: A caching algorithm generates a Markov process over the occupancy states of the cache. Each state C is a vector of size K indicating the content in each cache line, and a current $jump$ size. Each cache request generates a state transition based on the caching algorithm used via item entrances and evictions. Hence, for a given request arrival process, a caching algorithm is equivalent to a state transition matrix over the cache states. The typical performance analysis approach is then to determine the stationary distribution of the Markov process of occupancy states, and from it derive the hit probability.

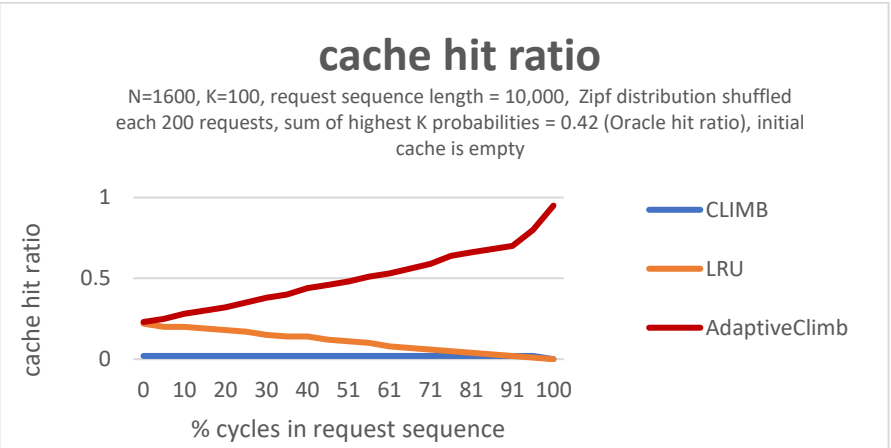
As an evidence of the quality of our algorithm, we theoretically calculated the expected hit ratio for $AdaptiveClimb$ versus LRU for various values of N, K, and requests' distribution settings, using Markov matrices. In all our calculations, $AdaptiveClimb$ achieves better results. Let's observe a result of such a calculation for $K = 3, N = 4, p_1 = 0.4, p_2 = 0.3, p_3 = 0.2, p_4 = 0.1$. We get an expected hit ratio of 0.833 for $AdaptiveClimb$, an expected hit ratio of 0.825 for LRU, and this leads to relative improvement of 1.009.

We test $AdaptiveClimb$ algorithm on real data which is taken from [EPA-HTTP]. The data contains 6620 different possible requests, a request sequence of length 47748, and a cache of size $K = 100$. $CLIMB$ hit ratio is 0.37, LRU hit ratio is 0.47, and $AdaptiveClimb$ hit ratio is 0.48.

There are situations where both LRU and $CLIMB$ perform very poorly. For example, consider the case where the sequence of requests is periodic, with a period exceeding the cache size. Obviously, both algorithms will achieve a 0% hit ratio. As another example, suppose that at some point, two items that are initially not in the cache, start appearing alternatingly. Employing $CLIMB$, after each request, we will insert to the cache the element just requested and evict the other. Again, we get a 0% hit ratio. $AdaptiveClimb$ algorithm may be a slightly modified to overcome these problems.

The modified $AdaptiveClimb$ algorithm uses two more parameters. The first indicates if the last request leaded to hit or miss. The second is a counter of consecutive cache misses that occurred while the value of $jump$ was maximal. When we get too many consecutive cache misses, the algorithm uses $jump = 1$ once, only for a single next request. After this the counter is set to zero. The following is an example where $AdaptiveClimb$ performs 2 times better than LRU, and much better than $CLIMB$.

We ran the algorithm with $N = 1600$ different possible requests, a cache of size $K = 100$, request sequence of length $10{,}000$ requests. Each request is selected using Zipf distribution. The distribution values of possible requests are randomly uniformly shuffled after each 200 requests. The sum of highest K probabilities is 0.42 which defines an optimal hit ratio for this experiment. The results of this experiment are presented in the Graph below. Even if the percentage of loop requests is 10% of the total number of requests, we get $AdaptiveClimb$ hit ratio of 0.35 versus LRU hit ratio of 0.18, and $CLIMB$ hit ratio of 0.02.



5. Future research directions

For our best knowledge, there is no theoretical proof that the expected hit ratio of $CLIMB$ algorithm is higher than expected hit ratio of LRU algorithm, but this is detected experimentally by previous work [CLIMB]. For both algorithms, the expected hit ratio depends on the cache size K and the number of possible requests N. Thus, it seems impossible to calculate the difference between these two formulas for a general case but only for concrete values of K and N. It may be interesting to prove this for general case. For this, one should look at a difference $P(CLIMB1_{hit}) - P(LRU_{hit})$, and try to prove that this expression is non-negative. Of course, it would be of high interest to calculate a general formula for expected hit ratio of $AdaptiveClimb$ algorithm by using the following formula to calculate hit ratio for LRU and $CLIMB$ algorithms, each according to its formula for steady-state probabilities:

$$P_{cache\ hit} = \sum_{i=1}^{N} (probability\ to\ request\ i) \cdot (probability\ that\ i\ is\ in\ cache)$$

$$= \sum_{i=1}^{N} p_i P(i \in cache) = \sum_{i=1}^{N} p_i \cdot \left(\frac{\sum_{\forall \overrightarrow{\sigma}:\, i \in \overrightarrow{\sigma}, ALG \in \{LRU, CLIMB\}} \pi_{(\overrightarrow{\sigma})}^{ALG}}{\sum_{i \in \overrightarrow{\sigma}, ALG \in \{LRU, CLIMB\}} \pi_{(\overrightarrow{\sigma})}^{ALG}} \right)$$

Acknowledgments

This research is partially supported by EMC.

Bibliography

[A-LRU] J. Li, S. Shakkottai, J. C. S. Lui, and V. Subramanian, "Accurate Learning or Fast Mixing? Dynamic Adaptability of Caching Algorithms," Arxiv preprint arXiv:1701.02214, 2017.

[LRU] E. G. Coffman and P. J. Denning, Operating Systems Theory. Prentice-Hall Englewood Cliffs, NJ, 1973.

[CLIMB] D. Starobinski and D. Tse, “Probabilistic Methods for Web Caching,” Performance evaluation, 2001.

[ACK87] O. Aven, E. Coman Jr., and Y. Kogan, Stochastic Analysis of Computer Storage, Series: Mathematics and its Applications, D. Reidel Publishing Company, 1987.

[EPA-HTTP] http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html

Detecting Malicious PowerShell Commands using Deep Neural Networks

Danny Hendler
Ben-Gurion University of the Negev
hendlerd@cs.bgu.ac.il

Shay Kels
Microsoft, Israel
shkels@microsoft.com

Amir Rubin
Ben-Gurion University of the Negev
amirrub@post.bgu.ac.il

ABSTRACT

Microsoft's PowerShell is a command-line shell and scripting language that is installed by default on Windows machines. Based on Microsoft's .NET framework, it includes an interface that allows programmers to access operating system services. While PowerShell can be configured by administrators for restricting access and reducing vulnerabilities, these restrictions can be bypassed. Moreover, PowerShell commands can be easily generated dynamically, executed from memory, encoded and obfuscated, thus making the logging and forensic analysis of code executed by PowerShell challenging.

For all these reasons, PowerShell is increasingly used by cybercriminals as part of their attacks' tool chain, mainly for downloading malicious contents and for lateral movement. Indeed, a recent comprehensive technical report by Symantec dedicated to PowerShell's abuse by cybercrimials [52] reported on a sharp increase in the number of malicious PowerShell samples they received and in the number of penetration tools and frameworks that use PowerShell. This highlights the urgent need of developing effective methods for detecting malicious PowerShell commands.

In this work, we address this challenge by implementing several novel detectors of malicious PowerShell commands and evaluating their performance. We implemented both "traditional" natural language processing (NLP) based detectors and detectors based on character-level convolutional neural networks (CNNs). Detectors' performance was evaluated using a large real-world dataset.

Our evaluation results show that, although our detectors (and especially the traditional NLP-based ones) individually yield high performance, an ensemble detector that combines an NLP-based classifier with a CNN-based classifier provides the best performance, since the latter classifier is able to detect malicious commands that succeed in evading the former. Our analysis of these evasive commands reveals that some obfuscation patterns automatically detected by the CNN classifier are intrinsically difficult to detect using the NLP techniques we applied.

Our detectors provide high recall values while maintaining a very low false positive rate, making us cautiously optimistic that they can be of practical value.

CCS CONCEPTS

• **Security and privacy** → **Domain-specific security and privacy architectures**; • **Computing methodologies** → **Neural networks**; *Supervised learning by classification*; *Ensemble methods*; • **Software and its engineering** → Scripting languages;

KEYWORDS

PowerShell, malware detection, neural networks, natural language processing, deep learning

ACM Reference Format:
Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting Malicious PowerShell Commands using Deep Neural Networks. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196494.3196511

1 INTRODUCTION

Modern society is more than ever dependent on digital technology, with vital sectors such as health-care, energy, transportation and banking relying on networks of digital computers to facilitate their operations. At the same time, stakes are high for cybercriminals and hackers to penetrate computer networks for stealthily manipulating victims' data, or wreaking havoc in their files and requesting ransom payments. Protecting the ever-growing attack surface from determined and resourceful attackers requires the development of effective, innovative and disruptive defense techniques.

One of the trends in modern cyber warfare is the reliance of attackers on general-purpose software tools that already preexist at the attacked machine. Microsoft PowerShell[1] is a command-line shell and scripting language that, due to its flexibility, powerful constructs and ability to execute scripts directly from the command-line, became a tool of choice for many attackers. Several open-source frameworks, such as PowerShell Empire[2] and PowerSploit[3] have been developed with the purpose of facilitating post-exploitation cyber-offence usage of PowerShell scripting.

While some work has been done on detecting malicious scripts such as JavaScript [9, 10, 30, 53], PowerShell, despite its prominent status in the cyber warfare, is relatively untreated by the academic community. Most of the work on PowerShell is done by security practitioners at companies such as Symantec [52] and Palo Alto Networks[37]. These publications focus mainly on surveying the

This research was partially supported by the Cyber Security Research Center and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196511

[1] https://docs.microsoft.com/en-us/powershell/
[2] https://www.powershellempire.com/
[3] https://github.com/PowerShellMafia/PowerSploit

PowerShell threat, rather than on developing and evaluating approaches for detecting malicious PowerShell activities. The discrepancy between the lack of research on automatic detection of malicious PowerShell commands and the high prevalence of PowerShell-based malicious cyber activities highlights the urgent need of developing effective methods for detecting this type of attacks.

Recent scientific achievements in machine learning in general, and deep learning [14] in particular, provide many opportunities for developing new state-of-the-art methods for effective cyber defense. Since PowerShell scripts contain textual data, it is natural to consider their analysis using various methods developed within the Natural Language Processing (NLP) community. Indeed, NLP techniques were applied for the sentiment analysis problem [31], as well as for the problem of detecting malicious non-PowerShell scripts [53]. However, adapting NLP techniques for detecting malicious scripts is not straightforward, since cyber attackers deliberately obfuscate their script commands for evading detection [52].

In the context of NLP sentiment analysis, deep learning methods considering text as a stream of characters have gained recent popularity and have been shown to outperform state of art methods [23, 55]. To the best of our knowledge, our work is the first to present an ML-based (and, more specifically, deep-learning based) detector of malicious PowerShell commands. Motivated by recent successes of character-level deep learning methods for NLP, we too take this approach, which is compelling in view of existing and future obfuscation attempts by attackers that may foil extraction of high-level features.

We develop and evaluate several ML-based methods for the detection of malicious PowerShell commands. These include detectors based on novel deep learning architectures such as Convolutional Neural Networks (CNNs) [13, 27] and Recurrent Neural Networks (RNNs) [12], as well as detectors based on more traditional NLP approaches such as linear classification on top of character n-grams and bag-of-words [32].

Detecting malicious PowerShell commands within the high volume of benign PowerShell commands used by administrators and developers is challenging. We validate and evaluate our detectors using a large dataset[4] consisting of 60,098 legitimate PowerShell commands executed by users in Microsoft's corporate network and of 5,819 malicious commands executed on virtual machines deliberately infected by various types of malware, as well as of 471 malicious commands obtained by other means, contributed by Microsoft security experts.

Contributions. The contributions of our work are two-fold. First, we address the important and yet under-researched problem of detecting malicious PowerShell commands. We present and evaluate the performance of several novel ML-based detectors and demonstrate their effectiveness on a large real-world dataset.

Secondly, we demonstrate the effectiveness of character-level deep learning techniques for the detection of malicious scripting. Our evaluation results establish that, although traditional NLP-based approaches yield high detection performance, ensemble learning that combines traditional NLP models with deep learning models further improves performance by detecting malicious commands that succeed in evading traditional NLP techniques.

[4]User sensitive data was anonymized.

Since the character-level deep learning approach is intrinsically language independent, we expect it can be easily adapted for detecting malicious usage of other scripting languages.

The rest of this paper is organized as follows. In Section 2, we provide background on PowerShell and how it is used as an attack vector and on some concepts required for understanding our deep-learning based detectors. In Section 3, we describe our dataset, how we pre-process commands and how our training set is constructed. A description of our detectors is provided in Section 4, followed by an evaluation of their performance in Section 5. Key related work is surveyed in Section 6. We conclude with a summary of our results and a short discussion of avenues for future work in Section 7. 8

2 BACKGROUND

2.1 PowerShell

Introduced by Microsoft in 2006, PowerShell is a highly flexible system shell and scripting technology used mainly for task automation and configuration management [7]. Based on the .NET framework, it includes two components: a command-line shell and a scripting language. It provides full access to critical Windows system functions such as the Windows Management Instrumentation (WMI) and the Component Object Model (COM) objects. Also, as it is compiled using .NET, it can access .NET assemblies and DLLs, allowing it to invoke DLL/assembly functions. These built-in functionalities give PowerShell many strong capabilities such as downloading content from remote locations, executing commands directly from memory, and accessing local registry keys and scheduled tasks. A detailed technical discussion of these capabilities can be found in [39].

As typical of scripting languages, PowerShell commands can be either executed directly via the command line, or as part of a script. PowerShell's functionality is greatly extended using thousands of 'cmdlets' (command-lets), which are basically modular and reusable scripts, each with its own designated functionality. Many cmdlets are built into the language (such as the `Get-Process` and `Invoke-Command` cmdlets), but additional cmdlets can be loaded from external modules to further enrich the programmer's capabilities. The `Get-Process` cmdlet, for instance, when given a name of a machine which can be accessed in the context in which PowerShell is executed, returns the list of processes that are running on that machine. As another example, the `Invoke-Command` cmdlet executes the command provided as its input either locally or on one or more remote computers, depending on arguments. The `Invoke-Expression` cmdlet provides similar functionality but also supports evaluating and running dynamically-generated commands.

2.1.1 PowerShell as an Attack Vector. While PowerShell can be configured and managed by the company IT department to restrict access and reduce vulnerabilities, these restrictions can be easily bypassed, as described by Symantec's comprehensive report about the increased use of PowerShell in attacks [52]. Furthermore, logging the code executed by PowerShell can be difficult. While logging the commands provided to PowerShell can be done by monitoring the shell that executes them, this does not necessarily provide the visibility required for detecting PowerShell-based attacks, since

PowerShell commands may use external modules and/or invoke commands using dynamically-defined environment variables.

For instance, the Kovter trojan [8] uses simple, randomly generated innocent-looking environment variables in order to invoke a malicious script. One such command that appears in our dataset is "`IEX $env:iu7Gt`", which invokes a malicious script referenced by the "iu7Gt" environment variable.[5] A log of the executing shell would only show the command before its dynamic interpretation, but will not provide any data regarding the malicious script.

Although Microsoft improved the logging capabilities of PowerShell 5.0 in Windows 10 by introducing the AntiMalware Scan Interface (AMSI) generic interface [6], many methods of bypassing it have already been published [41, 52], thus effective forensic analysis of malicious PowerShell scripts remains challenging.

In addition to the difficulty of forensic analysis, malware authors have several other good reasons for using PowerShell as part of their attacks [52]. First, since PowerShell is installed by default on all Windows machines, its strong functionality may be leveraged by cybercriminals, who often prefer using pre-installed tools for quicker development and for staying under the radar. Moreover, PowerShell is almost always whitelisted since it is benignly used by Windows system administrators [39].

Secondly, as PowerShell is able to download remote content and to execute commands directly from memory, it is a perfect tool for conducting file-less intrusions [22] in order to evade detection by conventional anti-malware tools. Finally, as we describe next, there are multiple easy ways in which PowerShell code can be obfuscated.

PowerShell Code Obfuscation. As described in [52], there are numerous ways of obfuscating PowerShell commands, many of which were implemented by Daniel Bohannon in 2016 and are publicly available in the "Invoke-Obfuscation" module he created [3]. Figure 1 lists a few key obfuscation methods we encountered in our data and provides examples of their usage. We now briefly explain each of them.

(1) As PowerShell commands are not case-sensitive, alternating lower and upper case letters often appear in malicious commands.
(2) Command flags may often be shortened to their prefixes. For instance, the "`-noprofile`" flag that excludes a PowerShell command from the execution policy can be shortened to "-nop".
(3) Commands may be executed using the "`-EncodeCommand`" switch. While the design goal for this feature was to provide a way of wrapping DOS-unfriendly commands, it is often used by malicious code for obfuscation.
(4) As mentioned previously, the "`Invoke-Command`" cmdlet evaluates a PowerShell expression represented by a string and can therefore be used for executing dynamically-generated commands.
(5) Characters can be represented by their ASCII values using "`[char]ASCII-VALUE`" and then concatenated to create a command or an operand.
(6) Commands may be base-64-encoded and then converted back to a string using the "`FromBase64String`" method.
(7) Base64 strings can be encoded/decoded in various ways (UTF8, ASCII, Unicode).
(8) Yet another way of obfuscating commands is to insert characters that are disregarded by PowerShell such as `.
(9) Command strings may be manipulated in real-time before evaluation using replacement and concatenation functions.
(10) The values of environment variables can be concatenated in run-time to generate a string whose content will be executed.
(11) Some malware generate environment variables with random names in every command execution.

While the ability to encode/represent commands in different ways and generate them dynamically at run-time provides for greater programming flexibility, Figure 1 illustrates that this flexibility can be easily misused. As observed by [52], "These [obfuscation] methods can be combined and applied recursively, generating scripts that are deeply obfuscated on the command line".

2.2 Deep Learning

In this section we provide background on deep learning concepts and architectures that is required for understanding the deep-learning based malicious PowerShell command detectors that we present in Section 4.

Artificial Neural Networks [47, 54] are a family of machine learning models inspired by biological neural networks, composed of a collection of inter-connected artificial *neurons*, organized in *layers*. A typical ANN is composed of a single *input layer*, a single *output layer*, and one or more *hidden layers*. When the network is used for classification, outputs typically quantify class probabilities. A *Deep Neural Network* (DNN) has multiple hidden layers. There are several key DNN architectures and the following subsections provide more details on those used by our detectors.

2.2.1 Convolutional Neural Networks (CNNs). A CNN is a learning architecture, traditionally used in computer vision [28, 29]. We proceed by providing a high-level description of the major components from which the CNN deep networks we use are composed.

As its name implies, the main component of a CNN is a *convolutional layer*. Assuming for simplicity that our input is a 2D grey scale image, a convolutional layer uses 2D $k \times k$ "filters" (a.k.a. "kernels"), for some integer k. As the filter is sliding over the 2D input matrix, the dot product between its $k \times k$ *weights* and the corresponding $k \times k$ window in the input is being computed. Intuitively, the filter slides over the input in order to search for the occurrences of some feature or patterm. Formally, given a $k \times k$ filter, for each $k \times k$ window x of the input to which the filter is applied, we calculate $w^T \cdot x + b$, where w is the filter's weights matrix and b is a *bias* vector representing the constant term of the computed linear function. The k^2 weights of w, as well as the k values of b, are being learnt during the training process.

Filters slide over the input in *strides*, whose size is specified in pixels. Performing the aforementioned computation for a single filter sliding over the entire input using stride s results in an output of dimensions $\big((n-k)/s+1\big) \times \big((n-k)/s+1\big)$, called the filter's "activation map". Using l filters and stacking their activation maps results in the full output of the convolutional layer, whose dimensions are $\big((n-k)/s+1\big) \times \big((n-k)/s+1\big) \times l$.

[5] `IEX` is an alias of `Invoke-Expression`.

ID	Description	Example
1	Using alternating lower and upper case letters	`-ExecUTIONPoLICy BypASs -wiNDoWSTYLe hidDeN (NEW-objecT SYstEM.NET.wEbCLIeNt).DOWnLoADFiLE(<removed>);`
2	Using short flags	`-nop -w hidden -e <removed>`
3	Using encoded commands	`-EncodedCommand <removed>`
4	Invoke expression using its string representation	`- Invoke-Expression (("New-Object Net.WebClient")).('Downloadfile') . . .`
5	Using "[char]" instead of a character	`. . . $cs = [char]71; $fn = $env:temp+$cs; . . .`
6	Reading data in base 64	`IEX $s=New-Object IO.MemoryStream([Convert]:: FromBase64String('<removed>'));`
7	Using UTF8 encoding	`$f=[System.Text.Encoding]::UTF8.GetString ([System.Convert]::FromBase64String(<removed>')); . . .`
8	Inserting characters overlooked by PowerShell like `	`. . . (new-object -ComObject wscript.shell).Popup(Ë-mail: <removed>@<removed>.com 'n 'nClient: <removed>"`) . . .`
9	String manipulation	`. . . $filename.Replace('-','/') . . . $env:temp + ':' + $name + '.exe . . .`
10	Concatenating variables inline	`$emnuxgy='i'; $jrywuzq='x'; $unogv='e';... Invoke-Expression ($emnuxgy+$unogv+$jrywuzq+' ' ...);`
11	Using a random name for a variable in every run	`iex $env:vruuyg`

Figure 1: Examples of PowerShell obfuscation methods.

In order to maintain the non-linear properties of the network when using multiple convolutional layers, a *non-linear layer* (a.k.a. *activation layer*) is added between each pair of convolutional layers. The non-linear layer applies a non-linear *activation function* such as the *Rectified Linear Units* (ReLU) function $f(x) = max(0, x)$ whose properties were investigated by [36] or the hyperbolic tangent $f(x) = tanh(x)$ function.

A *max pooling* layer [5] "down-samples" neurons in order to generalize and reduce overfitting [19]. It applies a $k \times k$ window across the input and outputs the maximum value within the window, thus reducing the number of parameters by a factor of k^2. A *fully connected layer* connects all inputs to all outputs. Intuitively, each output neuron of the convolutional layers represents an image feature. These features are often connected to the network's outputs via one or more fully connected layers, where the weights between inputs and outputs (learnt during the training process) determine the extent to which each feature is indicative of each output class.

Dropout layers [20] can be used between fully connected layers in order to probabilistically reduce overfitting. Given a probability parameter p, at each training stage, each node in the input remains in the network with probability p or is "dropped out" (and is disconnected from outputs) with probability $1 - p$. Dropout layers, as well as fully connected layers, may also apear in recurrent neural networks, described next.

2.2.2 Recurrent Neural Networks (RNNs). RNNs are neural networks able to process sequences of data representing, e.g., text [26, 34], speech [17, 18, 44], handwriting [15] or video [24] in a recurrent manner, that is, by repeatedly using the input seen so far in order to process new input. We use an RNN network composed of *long short-term memory* (LSTM) blocks [21]. Each such block consists of a *cell* that stores *hidden state*, able to aggregate/summarize inputs received over an extended period of time. In addition to the cell, an LSTM block contains 3 components called *gates* that control and regulate information flow into and out of the cell. Roughly speaking, the *input gate* determines the extent to which new input is used by the cell, the *forget gate* determines the extent to which the cell retains memory, and the *output gate* controls the level to which the cell's value is used to compute the block's output.

In the context of text analysis, a common practice is to add an *embedding layer* before the LSTM layer [42, 43]. Embedding layers serve two purposes. First, they reduce the dimensionality of the input. Secondly, they represent input in a manner that retains its context. The embedding layer converts each input token (typically a word or a character, depending on the problem at hand) to a vector representation. For example, when taking a character-level approach, one can expect that the representations of all digits computed by the embedding layer will be vectors that are close to each other. When the problem benefits from a word-level representation, *word2vec* [35] embeddings represent each word as a vector such that words that share common contexts in the text corpus using which the model was trained are represented by vectors that are close to each other.

A bidirectional RNN (BRNN) network [48] is an RNN architecture in which two RNN layers are connected to the output, one reading the input in order and the other reading it in reverse order. Intuitively, this allows the output to be computed based on information from both past and future states. BRNNs have found successful applications in various fields [2, 16, 51]. For instance, in the context of the sentiment analysis problem, when processing text from the middle of a sentence, text seen in the beginning of the sentence, as well as text seen at the end the sentence, may be used by the computation.

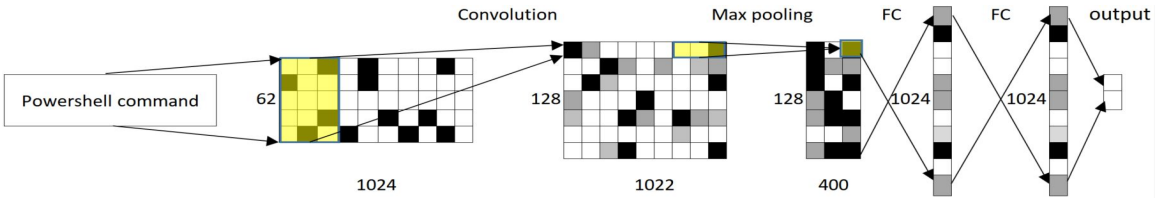


Figure 2: 4-CNN architecture used

3 THE DATASET

Our work is based on a large dataset which, after pre-processing (which we shortly describe), consists of 66,388 distinct PowerShell commands, 6,290 labeled as malicious and 60,098 labelled as clean. Malicious dataset commands belong to two types. For training and cross-validation, we use 5,819 distinct commands obtained by executing known malicious programs in a sandbox and recording all PowerShell commands executed by the program. For testing, we used 471 malicious PowerShell commands seen in the course of May 2017, contributed by Microsoft security experts. Using this latter type of malicious instances for evaluating our detection results mimics a realistic scenario, in which the detection model is trained using data generated inside a sandbox and is then applied to commands executed on regular machines.

As for clean commands, we received from Microsoft a collection of PowerShell commands executed within Microsoft's corporate network in the course of May 2017, on machines which had no indication of malware infection thirty days prior to the execution of the PowerShell command. Clean commands were split 48,094 for training and cross-validation and 12,004 for testing.

3.1 Pre-processing & Training Set Construction

We implemented a preprocessor whose key goals are to perform PowerShell command decoding and normalization for improved detection results. It also eliminates identical (as well as "almost identical") commands in order to reduce the probability of data leakage.

First, in order to be able to apply detection on "cleartext", our preprocessor decodes PowerShell commands that are encoded using base-64. Such commands are identified by the `-EncodedCommand` flag (or any prefix of it starting with '-e' or '-E'). All these commands undergo base-64 decoding, as otherwise they provide no useful detection data.[6]

Next, the preprocessor normalizes commands in order to reduce the probability of a data leakage problem [25] that, in our setting, may result from using almost-identical commands for training the model and for validating it. Indeed, we observed in our dataset PowerShell commands that differ only in a very small number of characters. In most cases, this was due to either the use of different IP addresses or to the use of different numbers/types of whitespace characters (e.g., spaces, tabs and newlines) in otherwise-identical commands. To avoid this problem, we replaced all numbers to asterisk signs ('*') and all contiguous sequences of whitespace characters to a single space and then eliminated duplicates.

We also observed in our dataset PowerShell *case-equivalent* commands that only differ in letter casing (see entry 1 in Figure 1). This was dealt with by ensuring that only a single command from each case-equivalence class is used for training/validation. We note that the dimensions of the dataset specified earlier relate to the numbers of distinct commands *after* this pre-processing stage.

Our dataset is very imbalanced, since the number of clean commands is an order of magnitude larger than that of malicious commands. In order to prevent model bias towards the larger class, we constructed the training set by duplicating each malicious command used for training 8 times so that the ratio of clean/malicious training commands is 1:1. We preferred to handle imbalance this way rather than by using under-sampling in order to avoid the risk of over-fitting, which may result when a neural network is trained using a small number of examples.

4 DETECTION MODELS

In this section we describe the machine learning models we used for malicious PowerShell command detection. We then evaluate and compare their performance in Section 5.

We implemented several deep-learning based detectors. In order to assess the extent to which they are able to compete with more traditional detection approaches, we also implemented detectors that are based on traditional NLP-based methods. We proceed by describing these two sets of models.

4.1 Deep-Learning Based Detectors

4.1.1 Input Preparation. Neural networks are optimized for classification tasks where inputs are given as raw signals [28, 29]. Using these networks for text classification requires to encode the text so that the network can process it. Zhang et al. [56] explored treating text as a "raw signal at character level" and applying to it a one-dimensional CNN for text classification. We take a similar approach for classifying PowerShell commands as either malicious or benign.

First, we select which characters to encode. We do this by counting for each character the number of training set commands in which it appears and then assigning a code only to characters that appear in at least 1.4% of these commands. We have set the encoding threshold to this value because at this point there is a sharp decline in character frequency. Thus, the least-frequent character encoded (which is `) appeared in approx 1.4% of commands and

[6] Command arguments encoded in either base-64 or UTF8 (see entries 6, 7 in Table 1) are not decoded since, in these cases, the encapsulating command is available and can be analyzed by the detector.

the most-frequent character that was not encoded (which is a non-English character) appeared in only approx 0.3% of the training set commands.

Rare characters are not assigned a code in order to reduce dimensionality and overfitting probability. The result is a set of 61 characters, containing the space symbol, all lower-case English letters (we soon explain how we represent upper-case letters) and the following symbols: `-'!%&()*,./:;?@[\]_` {|}+<=>û#$^~"`

Similarly to [56], we use input feature length of 1,024, so if a command is longer than that it is truncated. This reduces network dimensions and, as shown by our evaluation in Section 5.2, suffices to provide high-quality classification. The input to the CNN network is then prepared by using "one-hot" encoding of command characters, that is, by converting each character of the (possibly truncated) command to a vector all of whose first 61 entries are 0 except for the single entry corresponding to the character's code. All characters that were not assigned a code are skipped.

In practice, we use 62-long vectors rather than 61-long vectors in order to deal with the casing of English letters. Unlike in most NLP classification tasks, in the context of PowerShell commands character casing may be a strong signal (see obfuscation method 1 in Figure 1). In order to retain casing information in our encoding, we add a "case bit", which is the 62'nd vector entry. The bit is set to 1 if the character is an upper-case English letter and is set to 0 otherwise. Thus, the representation of a PowerShell command that is being input to the CNN network is a 62x1,024 sparse matrix. A matrix representing a command that is shorter than 1,024 is padded with an appropriate number of zero columns.

As we described in Section 2.2, whereas CNNs are traditionally used for computer vision and therefore typically receive as their input a matrix representing an image, recurrent neural networks (RNNs) are optimized for processing sequences of data. Consequently, the input we provide to our RNN classifier is a vector of numbers of size at most 1,024, whose i'th element is the code (as described above) of the i'th command character (characters that were not assigned a code are skipped), except that we explicitly encode upper-case English letters since we cannot use a case bit for the RNN input representation.

4.1.2 Training. Stochastic gradient descent is the most widely-used method for training deep learning models [4]. We train our deep-learning based algorithms using *mini-batch gradient descent*, in which each training *epoch* (a complete pass over the training set) is sub-divided to several *mini-batches* such that the gradient is computed (and network coefficients are updated accordingly) for each mini-batch.

In order to compare all our deep-learning networks on the same basis, in all our experiments we used 16 training epochs and mini-batch size of 128. We also experimented with other numbers of epochs/mini-batches but none of them obtained significantly better classification results.

4.1.3 Detection models. We implemented and evaluated 3 deep-learning based detectors described in the following.

(1) *A 9-layer CNN (9-CNN).* We use the network architecture designed by [56], consisting of 6 convolutional layers with stride 1, followed by 2 fully connected layers and the output layer. Two dropout layers are used between the 3 fully connected layers and a max pooling layer follows the first, second and last convolutional layers.[7] Unlike the architecture of [56] that uses fully connected layers of size 1,024 or 2,048, we use 256 entries in each such layer as this provides better performance on our data.
(2) *A 4-layer CNN (4-CNN).* We also implemented a shallower version of the 9-CNN architecture whose structure is depicted by Figure 2. It contains a single convolutional layer with 128 kernels of size 62x3 and stride 1, followed by a max pooling layer of size 3 with no overlap. This is followed by two fully-connected layers, both of size 1,024 – each followed by a dropout layer with probability of 0.5 (not shown in Figure 2), and an output layer.
(3) *LSTM.* We implemented a recurrent neural network model composed of LSTM blocks and used the character-level representation described above. Since inputs are not sentences of a natural language, we decided not to use Word2Vec [33] embedding. Instead, our LSTM architecture contains an embedding layer of size 32. The LSTM blocks we used are bi-directional LSTM cells with output dimension of 256, followed by two fully-connected layers, both of size 256, using a dropout probability of 0.5.

4.2 Traditional NLP-based detectors

We used two types of NLP feature extraction methods – a character level 3-gram and a bag of words (BoW). In both we evaluated both tf and tf-idf and then applied a logistic regression classifier on extracted features. The 3-gram model performed better using tf-idf, whereas BoW performed better using tf. For each detector we selected the hyper-parameters which gave the best cross-validation AUC results (evaluation results are presented in Section 5).

Note that as the 4-CNN architecture uses a kernel of length three in the first convolutional layer, the features it uses are similar to those extracted when using the character-level 3-gram detector.

4.3 Input Representation Considerations

Recalling the obfuscation methods used by PowerShell-base malware authors for avoiding detection (see Section 2.1.1), we observe that our input representation retains the information required for identifying them. The commands used for obfuscation, including their short versions (obfuscation method 2 in Figure 1), can be learnt due to the usage of 3-sized kernels by the deep-learning models and the usage of 3-grams by the traditional NLP models. Obfuscation method 3 is addressed by the decoding performed during data preparation (see Section 3.1).

Most other obfuscation methods (see Figure 1) use special characters such as "`", the pipe sign "|", the symbol "+" and the environment-variable sign "$". These special characters are often used when strings and the values of environment variables are concatenated in runtime for obfuscation. All these special characters appear in a significant fraction of our training set's commands and consequently they are all assigned codes by our input encoding for deep networks.

[7] Dropout and max pooling layers are typically not counted towards the network's depth.

Table 1: Detectors' area under the ROC curve (AUC) values.

9-CNN	4-CNN	4-CNN*	LSTM	3-gram	BoW
0.985	0.988	0.987	0.988	0.990	0.989

They are also retained in the input provided to the traditional NLP models.

As for the usage of random names (obfuscation method 11), these typically include numbers (converted to the '*' sign) or alternating casing, and can therefore be learnt by our classifiers as well. (As we describe later, our deep learning classifiers do a better job in learning such patterns.) The usage of special strings such as "[char]", "UTF8", "Base64" or the character '`' is also covered by both models as they are retained in the input.

The only obfuscation method w.r.t. which the input to some of our detectors is superior to that provided to others is the usage of alternating lower/upper case characters (obfuscation method 1 in Figure 1). Whereas the case-bit was easily incorporated in the input to our CNN deep-learning classifiers, the RNN and the traditional NLP-based models input representations do not accommodate its usage.

5 EVALUATION

We performed 2-fold cross validation on the training data and present the area under the ROC curve (AUC) results (rounded to the third decimal place) of our detectors in Table 1. In addition to the 5 detectors presented in Section 4, we also evaluated a variant of 4-CNN (denoted 4-CNN*) in which we did not use the case bit.

All detectors obtain very high AUC levels in the range 0.985 – 0.990. The traditional NLP-based detectors provide excellent results in the range 0.989 – 0.990, the 4-CNN and LSTM detectors slightly lag behind with AUC of 0.988 and 9-CNN provides a lower AUC of 0.985. The 4-CNN* detector provides slightly lower AUC than that of 4-CNN, establishing that the case bit is beneficial.

For a detector to be practical, it must not produce many false alarms. As the cyber security domain is often characterized by a very high rate of events requiring classification, even a low false-positive rate (FPR) of (say) 1% may result in too many false alarms. It is therefore important to evaluate the true positive rate (TPR) (a.k.a. *recall*) provided by detectors when their threshold is set for low FPR levels.

Table 2 presents the TPR of our detectors for FPR levels 10^{-2}, 10^{-3} and 10^{-4} on both the training/cross-validation and the test sets. Since we have a total of about 12,000 clean commands in the test set, we stop the analysis at FPR level of 10^{-4}. Presented values in the "Cross-validation" part of the table are the average of the two folds. Values in the "Test set" part were obtained by models trained on the training set in its entirety.

Focusing first on cross-validation results, it can be seen that, while all classifiers achieve high TPR values even for very low FPR levels, the performance of the traditional NLP detectors is better. The 3-gram detector leads in all FPR levels with a gap that increases when FPR values are decreased. Specifically, even for an FPR of 1:10,000 it provides an excellent TPR of 0.95. Among the deep-learning based detectors, 4-CNN and LSTM are superior to 4-CNN* and 9-CNN. For FPR rate of 1:10,000, 4-CNN and LSTM provide TPRs of 0.89 and 0.85, respectively. 9-CNN obtains the worst results in all experiments.

Results on the test set are significantly lower but still good. It is noteworthy that the gaps between the traditional NLP and the 4-CNN/LSTM models that we observed on the training data almost vanish on the test data. This seems to indicate that the latter models are able to generalize better.

For an FPR of 1:100, the best performers are 4-CNN and 4-CNN* with a TPR of 0.89, LSTM is second best with 0.88 and both the 3-gram and BoW detectors obtain a TPR of 0.87. For FPR 1:1,000 the 3-gram detector is best with TPR of 0.83, only slightly better than LSTM's 0.81 TPR, and for FPR 1:10,000, all of 3-gram, 4-CNN and LSTM (ordered in decreasing performance) identify approximately two thirds of malicious commands. The significance of the case bit is evident when comparing the results of the 4-CNN and the 4-CNN* detectors on the test set for FPR level of 1:10,000. The TPR when using the case bit (4-CNN) is higher by almost one third than that when it is not used (4-CNN*). 9-CNN is the worst performer also in the test set experiments, by a wider margin than in the cross-validation tests.

As we've mentioned, the performance on the test set is significantly lower than that of cross-validation in all experiments. This is to be expected: whereas training set malicious commands were generated by running malware inside a sandbox, the malicious commands in the test set were contributed by security experts. Consequently, test set malicious commands may have been collected in different ways (e.g. by searching the Windows registry for malicious PowerShell commands) and may have been produced by malware none of whose commands are in the training set.

5.1 A Deep/Traditional Models Ensemble

We next show that by combining 4-CNN – our best deep learning model and 3-gram – our best traditional NLP model, we are able to obtain detection results that are better than those of each of them separately. We then analyze the type of malicious commands for which the deep model contributes to the traditional NLP one.

The *D/T Ensemble* is constructed as follows. We classify a command using both the 4-CNN and the 3-gram detectors, thus receiving two scores. If either one of the scores is 0.99 or higher, we take the maximum score, otherwise we take the average of the two scores. We evaluated the Ensemble's TPR by FPR performance on the test set in the same manner we evaluated the non-Ensemble

Table 2: TPR by FPR per model: cross-validation and test results.

FPR	Cross-validation			Test set		
	10^{-2}	10^{-3}	10^{-4}	10^{-2}	10^{-3}	10^{-4}
9-CNN	0.95	0.89	0.73	0.72	0.52	0.24
4-CNN	0.98	0.96	0.89	0.89	0.76	0.65
4-CNN*	0.97	0.93	0.85	0.89	0.72	0.49
LSTM	0.98	0.95	0.85	0.88	0.81	0.64
3-gram	0.99	0.98	0.95	0.87	0.83	0.66
BoW	0.98	0.93	0.87	0.87	0.50	0.35

algorithms (see Table 2). The D/T Ensemble significantly outperformed all non-Ensemble algorithms and obtained on the test set TPRs of 0.92, 0.89 and 0.72 for FPR levels of 1:100, 1:1,000 and 1:10,000, respectively.

In order to gain better visibility into the contribution of the 4-CNN detector on top of the 3-gram detector, we present in Figures 3a-3c the confusion matrixes of the 3-gram, 4-CNN and D/T Ensemble detectors on the test set. These results are obtained using the lowest threshold (for each of the algorithms) that provides an FPR of no more than 10^{-3}. Since the test set contains approximately 12,000 clean instances, this means that the algorithms must have at most 12 false positives.

As can be seen by comparing Figures 3a and 3c, the D/T Ensemble adds 42 new detections on top of those made by the 3-gram detector, with only 4 new false positives. We analyzed these new detections in order to understand where the deep learning model is able to improve over the traditional NLP model.

Out of the new 42 detected commands, 15 commands contain a sequence of alternating digits and characters. In most cases, this sequence represented the name of the host or domain from which the command downloaded (most probably malicious) content. Recall that in our pre-processing of commands, we convert digits to asterisks (see Section 3.1), thus the host/domain name contains many asterisks in it. An example of the usage of such a name that appeared in one of the newly detected commands is:
"`...DownloadFile('http://d*c*a*ci*x*.<domain>')...`".

Each of these names appears only once and they are most probably generated by a domain generation algorithm (DGA) [50] used by the malware for communicating with its command and control center. Since these names are unique and seem random, the 3-gram algorithm is unable to learn their pattern, while the neural network is able to.

Figure 4a depicts an example of how such a host name is encoded in the input to the neural network. Note the pattern of alternating zeros and ones in the row corresponding to the symbol '*'. Figure 4b depicts a neural network filter of size 3 that is able to detect occurrences of this pattern. The filter contains ones in the first and third columns of the row corresponding to '*' (where the '*' symbol is expected to be found) and a zero in the middle column of that row, signifying that the character between the two digits is of no significance. When this filter is applied to the characters sequence depicted in Figure 4a, it creates a relatively strong signal. On the other hand, considering the 3-gram's feature extraction algorithm, since the character between the two digits changes from one command to the other, the model is unable to learn this pattern.

A similar argument can explain the detection of a few additional commands by the D/T Ensemble that were not detected by 3-gram. These commands contain a random sequence of characters alternating between lower and upper case, most probably generated by a DGA algorithm as well. Using the case bit provided as part of its input, 4-CNN is able to identify this pattern.

We note that in both the above cases, the PoweShell commands may include additional indications of maliciousness such as the web client or the cmdlets they use. Nevertheless, it is the ability to detect patterns that incorporate random characters and/or casing that causes 4-CNN to assign these command a score above the threshold, unlike the 3-gram detector.

Our ensemble detector had only seven false positive (FPs), which we manually inspected. Two FPs exhibited obfuscation patterns – one used `[System.Text.Encoding]::UTF8.GetString` (usage of `UTF8` was observed in 1,114 of the clean commands) and the other used the `-EncodedCommand` flag (which was observed in 1,655 of the clean commands). The remaining five FPs did not use any form of obfuscation, but they all used at least two flags such as `-NoProfile` and `-NonInteractive` (each seen in 5,014 and 5,833 of the clean commands, respectively).

5.2 Command Length Considerations

As previously mentioned, our detectors receive as input a 1,024-long prefix of the PowerShell command and longer commands are being truncated. As shown by our evaluation, this suffices to provide high-quality classification on our dataset.

A possible counter-measure that may be attempted by future malware authors for evading our detection approach is to construct long PowerShell commands such that malicious operations only appear after a long innocent-looking prefix consisting of harmless operations. In the following, we explain how such a hypothetic counter-measure can be thwarted.

Analyzing our dataset's distribution of command lengths, we find that the length of 86% of all malicious commands and 88% of all clean commands is 1,024 or less. Moreover, the length of 96.7% of all malware commands and, more importantly, *the length of 99.6% of all clean commands is 2000 or less*. We remind the reader that all commands were used by our detectors regardless of their length – commands longer than 1,024 characters were simply truncated. Given the good performance of all detectors, we found no reason of using a longer input size. It would be straightforward to modify our detectors for accommodating inputs of size 2,048 or longer if and when the characteristics of malicious commands change such that this would be necessary. As of now, clean commands whose length exceeds 2000 are very rare, deeming them suspicious.

Figure 5 presents the command-length distributions of benign and malicious commands in our dataset for commands of length 1,024 or less. The distribution of malicious command length is relatively skewed to the right, indicating that malicious PowerShell commands tend to be longer than benign commands. The high peak of very short malicious commands is to due to Kovter trojan commands [8] that constitute approximately 8% of the malicious commands population in our dataset.

6 RELATED WORK

Zhang et al. [56] introduced a deep-learning approach for text classification in which the input to convolutional neural networks (CNNs) is at character-level instead of word-level. They compared their deep-learning based classifiers with word-based traditional NLP methods (such as n-grams) and with recurrent neural networks (using LSTM blocks). Their empirical evaluation was conducted using sentiment analysis and topic classification datasets. Their results show that, while traditional methods provided better performance on small/medium datasets, character-based CNNs outperformed them on larger datasets. Our 9-CNN architecture is almost identical to theirs and its inputs are encoded in a similar manner.

prediction outcome

actual value	p	n
p′	373	98
n′	3	12001

(a) 3-gram

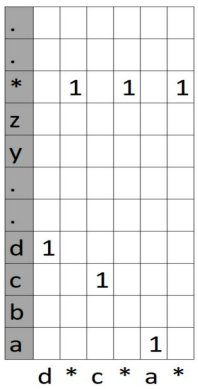
actual value	p	n
p′	340	131
n′	5	11999

(b) 4-CNN

actual value	p	n	total
p′	415	56	471
n′	7	11997	12004

(c) D/T Ensemble

Figure 3: Confusion matrices for 3-gram, 4-CNN and Ensemble on test set, using thresholds resulting in FPR lower than 10^{-3}.



(a) A sample hostname encoding (zeros removed for clarity).

.	0	0	0
.	0	0	0
*	1	0	1
z	0	0	0
y	0	0	0
.	0	0	0
.	0	0	0
d	0	0	0
c	0	0	0
b	0	0	0
a	0	0	0

(b) Filter capable of identifying alternating digits.

Figure 4: A hostname encoding and a filter which was used by the network to identify alternating digits and letters

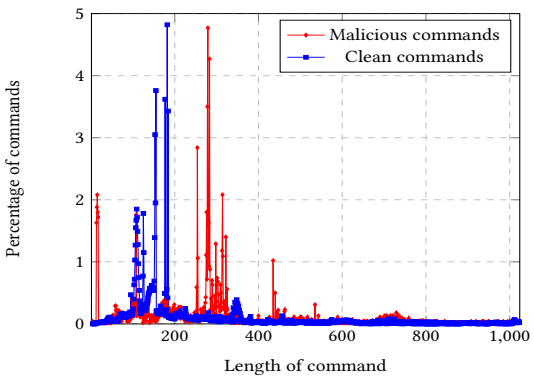


Figure 5: PowerShell command-length distributions of clean vs malicious commands.

Prusa and Khoshgoftaar [40] compare several architectures for short text sentiment analysis classification applied on a large dataset of tweets. They show that two relatively shallow architectures (one comprising 3 convolutional layers and 2 fully connected layers and the other comprising a single convolutional layer followed by a single LSTM layer) gave the best results. Our results are aligned with theirs in that also in our empirical evaluation the relatively shallow 4-CNN network achieved better classification performance than the deeper 9-CNN network. In both settings, classified text is relatively short – up to 140 characters inputs in their study and up to 1,024 characters in ours.

Deep learning approaches are increasingly used in recent years for malware detection. Some of these works (see [1, 11, 38, 45] for a few examples) classify programs as either malicious or benign based on their binary code and/or their runtime behaviour. In order for the neural network to be able to classify executable programs, a non-trivial feature extraction pre-processing stage is typically required whose output is fed to the neural network.

Athiwaratkun and Stokes [1] used a large dataset consisting of Windows portable executable (PE) files. They applied deep models to inputs representing the system calls made by these programs. They implemented and evaluated several models, including a character-level CNN similar to the one used by [56]. Unlike our results, in their empirical evaluation the LSTM model achieved the best results. However, none of their neural networks was shallow.

Smith et al. also studied the problem of malware detection based on system calls made by PE executables [49]. They used several classification algorithms, including Random Forest, CNN and RNN. They observed a decay in classification quality when input length exceeded 1,000 system calls. Although problem setting and input domains differ, both our work and theirs provide evidence that limiting input length by some (domain specific) threshold may be sufficient (and is sometimes even required) for obtaining good performance.

Similarly to our work, Saxe and Berlin use deep learning models for malware detection by analyzing "cleartext" [46]. More specifically, they apply these models on a large dataset consisting of (both benign and malicious) URLs, file paths and registry keys. Their CNN architecture uses a single convolutional layer, as does our 4-CNN model.

Although some previous studies investigated the problem of detecting malicious scripting-language commands/scripts (where cleartext classification can be applied), to the best of our knowledge none of them addressed PowerShell. Several prior works presented detectors of malicious JavaScript commands by employing feature extraction pre-processing followed by the application of a shallow classifier (see, e.g., [9, 10, 30]).

Wang et al. used deep models for classifying JavaScript code collected from web pages [53]. Similarly to our work, their model uses character-level encoding, with an 8-bit character representation. They compare their classifiers with classic feature extraction based methods and study the impact of the number of hidden layers and their size on detection accuracy.

A few reports by AV vendors published in recent years surveyed and highlighted the potential abuse of PowerShell as a cyber attack vector [37, 39, 52]. Pontiroli and Martinez analyze technical aspects of malicious PowerShell code [39]. Using real-world examples, they demonstrate how PowerShell and .NET can be used by different types of malware. Quoting from their report: "Vast amounts of ready-to-use functionality make the combination of .NET and PowerShell a deadly tool in the hands of cybercriminals".

A recent comprehensive technical report by Symantec dedicated to PowerShell's abuse by cybercrimials [52] reported on a sharp increase in the number of malicious PowerShell samples they received and in the number of penetration tools and frameworks that use PowerShell. They also describe the many ways in which PowerShell commands can be obfuscated.

Collectively, these reports shed light on the manner in which PowerShell can be used in different stages of a cyber attacks – from downloading malicious content, through reconnaissance and malware persistence, to lateral movement attempts. We have used a few of the insights they provide on PowerShell attacks for designing our detection models and for preprocessing PowerShell commands.

As we've mentioned previously, Microsoft improved the logging capabilities of PowerShell 5.0 in Windows 10, with the introduction of the AntiMalware Scan Interface (AMSI), but many methods of bypassing it have already been published. This problem was discussed and addressed in [41], where the fact that PowerShell is built on .NET architecture was used for monitoring PowerShell's activity, by leveraging .NET capabilities. As discussed in their work, the proposed solutions require some adjustments which may hurt PowerShell's performance, as well as generate some artifacts on the machine.

7 DISCUSSION

PowerShell commands can be executed from memory, hence identifying malicious commands and blocking them prior to their execution is, in general, impractical. We therefore estimate that the most plausible deployment scenario of our detector would be as a post-breach tool. In such a deployment scenario, PowerShell commands that execute will be recorded and then classified by our detector. Commands classified as malicious would generate alerts that should trigger further investigation. In corporate networks, this type of alerts is typically sent to a security information and event management (SIEM) system and presented on a dashboard monitored by the organization's CISO (chief information security officer) team.

There are several ways in which this work can be extended. First, while we have implemented and evaluated several deep-learning and traditional NLP based classifiers, the design space of both types of models is very large and a more comprehensive evaluation of additional techniques and architectures may yield even better detection results.

Secondly, in this work we targeted the detection of individual PowerShell commands that are executed via the command-line. An interesting direction for future work is to devise detectors for complete PowerShell scripts rather than individual commands. Such scripts are typically longer than single commands and their structure is richer, as they generally contain multiple commands, functions and definitions. Effective detection of malicious scripts would probably require significantly different input encoding and/or detection models than those we used in this work.

Another interesting avenue for future work is to devise detectors that leverage the information collected by Microsoft's AntiMalware Scan Interface (AMSI) [6]. As mentioned previously, AMSI is able to record PowerShell commands (generated both statically and dynamically) that are executed in run-time, so detectors may have more data to operate on. However, although AMSI may be less vulnerable to many of the obfuscation methods described in Section 2.1.1, attackers may be able to find new ways of camouflaging the AMSI traces of their malicious commands.

8 CONCLUSION

In this work we developed and evaluated two types of ML-based detectors of malicious PowerShell commands. Detectors based on deep learning architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), as well as detectors based on more traditional NLP approaches such as linear classification on top of character n-grams and bag-of-words.

We evaluated our detectors using a large dataset consisting of legitimate PowerShell commands executed by users in Microsoft's corporate network, malicious commands executed on virtual machines deliberately infected by various types of malware, and malicious commands contributed by Microsoft security experts.

Our evaluation results show that our detectors yield high performance. The best performance is provided by an ensemble detector that combines a traditional NLP-based classifier with a CNN-based classifier. Our analysis of malicious commands that are able to evade the traditional NLP-based classifier but are detected by the CNN classifier reveals that some obfuscation patterns automatically detected by the latter are intrinsically difficult to detect using traditional NLP-based classifiers. Our ensemble detector provides high recall values while maintaining a very low false positive rate and so holds the potential of being useful in practice.

REFERENCES

[1] Ben Athiwaratkun and Jack W Stokes. 2017. Malware classification with LSTM and GRU language models and a character-level CNN. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, ., 2482–2486.

[2] Pierre Baldi, Søren Brunak, Paolo Frasconi, Giovanni Soda, and Gianluca Pollastri. 1999. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics* 15, 11 (1999), 937–946.

[3] Daniel Bohannon. 2016. The Invoke-Obfuscation module. https://github.com/danielbohannon/Invoke-Obfuscation. (2016).

[4] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, ., 177–186.

[5] Y-Lan Boureau, Francis Bach, Yann LeCun, and Jean Ponce. 2010. Learning mid-level features for recognition. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, ., 2559–2566.

[6] Microsoft Corporation. 2017. Antimalware Scan Interface. https://msdn.microsoft.com/he-il/library/windows/desktop/dn889587(v=vs.85).aspx. (2017).

[7] Microsoft Corporation. 2017. PowerShell. https://docs.microsoft.com/en-us/powershell/scripting/powershell-scripting?view=powershell-5.1. (2017).

Design of Nano-Robots for Exposing Cancer Cells

Shlomi Dolev[1], Michael Rosenblit[2] and Ram Prasadh Narayanan[1]

[1]Department of Computer Science, Ben-Gurion University of the Negev. Email: {dolev,narayan}@cs.bgu.ac.il
[2]Ilse Katz Institute of Nano-Scale Science and Technology, Ben-Gurion University of the Negev. Email: rmichael@bgu.ac.il

May 29, 2018

Abstract

A proof-of-concept design of a nano-robot which can navigate, detect cancer cells and actuate the release of chemicals in blood is discussed. The nano-robot is designed with blood energy harvesting capability and accumulation of electricity in a capacitor, that forms the main body of the nano-robot. The nano-robot is immobilized with glucose hunger-based cancer detectors that reduces the electrical resistance of a nano-tube, when attached to a cancer cell. This mechanism, in-turn allows electric current to activate a nano-electrical-mechanical (NEM) relay (mechanical transistor) to break, exposing a chemical material identified by the immune system for cell elimination. This concept is in line with the effort to design an autonomous computational nano-robot for in-vivo medical diagnosis and treatment. The concept can also be considered as a step to bridge the gap between theoretical swarming/navigation techniques and a computational hardware for plausible implementation of the theory.

1 Introduction

In general, programmable matter is any matter that has the ability to change its physical properties (like shape, density, moduli, conductivity, optical properties, etc.) based on user input or autonomous sensing. We are particularly interested in implementable programmable matter, composed of nano-robots. Key applications to nano-robotics are medical target identification, targeted drug delivery and minimal invasive surgery [1–3], to name a few. In our previous work [4], we presented a swarming algorithm for *in-vivo* nano-robots inspired from caterpillar swarm in nature, wherein, we suggested instructing the computational particles to create layered and connected structures, for the benefit of speed and energy preservation. We put forward a proposition that the theoretical concept needs to be supported by hardware feasibility for a plausible implementation of a computational nano-robot.

Through this work, we have attempted to design autonomous nano-robots which can harvest energy from the glucose in the blood and activate a response based on bio-detection. We will present below the design of the nano-robot structure with three modules, (*i*) External coated energy harvester electrodes and cylindrical capacitor, (*ii*) Detector and (*iii*) Actuator. We believe this facile theory of inorganic nano-robot platform design can help to bridge the gap between existing research on bio-nano sensing in combination with advanced nano transistor technology towards cancer treatment. A collective system of electrical manipulation, bio-detection and NEM actuation can visualize the programmability in the nano matters.

2 Design of Nano-Robot Structure

2.1 Energy harvester and Metal-Insulator-Metal (M-I-M) cylindrical nano-capacitor

A nano-robot can be termed autonomous if there is no dependency on external sources for its operation. The first step to visualize an autonomous nano-robot is to provide an energy source to it. Thus, as a first module, we wanted to show that autonomy can be introduced to the nano-robot by harvesting energy from blood. At injection, the nano-robots are under constant influence of the blood. The velocity of blood flow ranges from $0.3 - 40$ cm/sec and the glucose concentration in blood is $(3.9 - 6.4)$ mM/L. With an estimated harvester total surface area of 2.041×10^{-10} cm^2 and with the displacement of $0.3 - 40$ cm, the volume of blood interacting with the nano-robot per second is estimated to be $(6.123 - 816.4) \times 10^{-14}$ L/Sec. On average, if 5.5 mM/L is the concentration of glucose, the number of moles of glucose interacting with the nano-robot and the number of electrons in the interacting glucose concentration is calculated to be $(33.67 - 4490.2) \times 10^{-14}$ mM/Sec and $(40.4 - 5408.8) \times 10^{10}$ $Electrons/Sec$, respectively. This gives the source current of the harvester to be $(64.64 - 8654)$ nA. The capacitance is calculated to be 4.702×10^{-16} F[1].

To compliment the calculations of the capacitance, we simulated a 3-D model of a concentric cylindrical Metal-Insulator-Metal (M-I-M) encapsulated capacitor using COMSOL Multiphysics (License #17076110). The simulated structure is precisely maintained within a $100nm$ dimension to (i) qualify the structure as a nano-robot and (ii) to allow the nano-robot to immobilize itself on the cell surface and act as a parasite. The provided energy harvesting capability will enhance the nano-robot to power, in particular, the detector to be used for actuation.

2.2 Bio-Detector

The second part of our proof-of-concept is to show the bio-detection mechanism of cancer cells and to provide an electrical output to logical decisioning. Detection of tumor cells, *in-vivo*, also means an active navigation of the nano-robots using chemical pheromones, acoustic vibrations, photo/fluorophores etc., for binding to tumor tissues. In our case, the glucose hunger of cancer cells [5] and the nano-robots immobilized with glucose or its variants, yields their navigation in blood. Assuming now, that our nano-robots can move swim randomly in blood, harvest and store energy from the blood glucose, the next step would be to utilize stored charges for a logical decision making. The black-box between harvested energy and decisioning is bio-detection.

A chemical detection of cancer cell should induce a change in the charge flow between the electrodes. This hysteresis in charge flow rate can be considered as a normalized electrical low and high. The change in specific conductance of the detector is considered as the electrical signature for detection of the cancer. Details on the design of detector and its electrical output parameters are explained in the technical report.

2.3 Actuator

Until the previous sections, the concept is extended to the point where the nano-robots are autonomous in terms of their power and navigation and computationally equipped to handle a bit change of information. In this section, we will discuss the possibility of a Nano-Elecro-Mechanical (NEM) switch which gets triggered by the conductance change of immobilized CNT bio-detector. This will ensure that the matters will respond appropriately depending on the decided logical operation. Miniature mechanical switches suitable for applications which require $sub - 100mV$ operation was demonstrated here [6, 7]. We show that a mechanical chamber can be designed to lock necessary drug or a combination of drugs, that can be unlocked, when increase in electrical

[1] **A detailed technical report with all the calculations and simulations, is linked here (TR 18-02).**

power flow due to bio-detection provides adequate electrostatic force to break the ceiling of the chamber, eventually exposing the drug to the environment. This signal is expected to cause a mechanical break, high enough to surpass the stress gradient of the ceiling structure in the actuator. The mechanical break exposes a detectable drug causing the immune system to recognize the attached nano-robot and the cancer cell. The electrostatic force F_e acting on the parallel plates due to change in dielectric thickness is given by

$$F_b = F_e = \frac{1}{2} V_b^2 \frac{\epsilon A_c}{d^2} \tag{1}$$

where V_b and F_b are the breakdown voltage and force of the NEM chamber with the glued drug inside it and A_c is the area of the ceiling and d is the distance between the plates. The resistances R_1 and R_2 tunes the breakdown voltage V_b, such that the NEM chamber collapses. An increase in input voltage, V_{in} due to detection, will result in the breakage of NEM box (when V_{in} reaches V_b), thus exposing the glued drug. The electrostatic force F_e on the chamber electrode was found to be $\approx 1.476nN$. It is presumed that the electrostatic force acting on the chamber surface is smaller than (before detection) and higher than (after detection) $F_{e_{min}}$ and $F_{e_{max}}$ respectively, i.e,

Before Detection: $F_s = \frac{1}{2} V_s^2 \frac{\epsilon A_c}{d^2}$, where, $F_s < F_{e_{min}}$ (*Stable condition*)

After Detection: $F_b = \frac{1}{2} V_b^2 \frac{\epsilon A_c}{d^2}$, where, $F_b \geq (F_{e_{max}} + \Delta F_e)$ (*Breaking condition*)

3 Conclusion and future work

A facile approach of an energy harvester, glucose hunger detector for cancer cells and actuator for drug exposure design is discussed. A collective system of electrical manipulation, bio-detection and NEM actuation to visualize the programmability in nano-robots is presented. The calculations and simulation results provide a proof-of-concept towards a plausible implementation of an autonomous computational nano-robot with bio-detection, logical decisioning and actuation for drug exposure. To the best of our knowledge, this work is the first of its kind that presents an overall picture towards an autonomous computational nano-robot for cancer diagnosis and treatment.

Acknowledgments

We thank the Lynne and William Frankel Center for Computer Science, the Rita Altura Trust Chair in Computer Science, the Krietman School of Advanced Graduate Studies, Ben-Gurion University of the Negev for their support to this work. We also thank Prof. Zeev Zalevsky from Bar-Ilan University for his useful comments and assistance.

References

[1] Wikipedia. Size of a human cell.

[2] P. Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440:297–302, 2006.

[3] C. Montemagno and G. Bachand. Constructing nanomechanical devices powered by biomolecular motors. *Nanotechnology*, 10:225–231, 1999.

[4] S. Dolev, S. Frenkel, M. Rosenblit, R. P. Narayanan, and K. M. Venkateswarlu. In-vivo energy harvesting nano robots. In *2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE)*, pages 1–5, Nov 2016.

[5] M. V. Liberti and J. W. Locasale. The warburg effect: how does it benefit cancer cells? *Trends in biochemical sciences*, 41(3):211–218, 2016.

[6] B. Osoba, B. Saha, L. Dougherty, J. Edgington, C. Qian, F. Niroui, J. H. Lang, V. Bulovic, J. Wu, and T. J. K. Liu. Sub-50 mv nem relay operation enabled by self-assembled molecular coating. pages 26.8.1–26.8.4, Dec 2016.

[7] C. Qian, A. Peschot, B. Osoba, Z. A. Ye, and T. J. K. Liu. Sub-100 mV computing with electro-mechanical relays. *IEEE Transactions on Electron Devices*, 64(3):1315–1321, 2017.

Learning Temporal Latent Variable Models

Dan Halbersberg and Boaz Lerner
halbersb@post.bgu.ac.il and boaz@bgu.ac.il

Ben-Gurion University of the Negev

Abstract. Latent variables may represent the origin of almost any scientific, social, or medical phenomena. While models with only observed variables have been well studied, learning a latent variable model (LVM) allowing both types of variables is more difficult. Moreover, latent variables by nature often change dynamically (temporally), which adds to the challenge and complexity of learning, but current LVM learning algorithms do not account for the natural dynamics. Here, we propose local learning of an LVM based on pairwise cluster comparison to learn intra- and inter-slice edges of a dynamic Bayesian network with latent variables and causal interrelationships among latent variables, in addition to those with observed variables.

Keywords: latent variable models · temporal models · causal discovery.

1 Introduction

Our world is very complex and sophisticated composed of variables that are observed and unobserved (latent). Latent variables can be psychological concepts such as mood and satisfaction, or sensory but difficult or expensive to measure (e.g., quarks, some medical tests). In learning an LVM, we do not know their number, cardinality (number of states), interrelations, and relations to the observed variables that measure them. Finding causal relations manifested in an LVM is a most difficult problem because, in general, the joint distribution can be generated by an infinite number of different LVMs [4]. Structural EM (SEM) learns an LVM given the number of latent variables and their cardinality as an input. The fast-causal inference (FCI) algorithm finds evidence for the existence of latent variables, but it does not explicitly finds them. The hierarchical latent class (HLC) algorithm learns a rooted tree in which all non-leaf nodes are latent variables, however, since it aims at the untrue assumption that observed variables are independent given the class variable, it does not provide casual explanations. A subclass of LVM is the multiple indicator model (MIM). BuildPureClusters (BPC) [4] learns an equivalence class of a MIM by searching for the set of vanishing tetrad constraints. Learning pairwise cluster comparison (LPCC) [2] leverages cluster analysis to identify from instantiations of the observed variables the latent ancestors of these variables. The main idea is that any change in the value of an exogenous (EX) (i.e., zero in-degree) latent variable should be reflected in the values taken by the non-EX latent and observed variables that are descendants of this EX. Thus, if we want to identify an EX latent variable, we should check for changes in the observed variables that are its descendants. These can be revealed by comparing data clusters characteristics over observed variable instantiations, e.g., by comparing cluster centroids pairwise; each comparison reveals a value configuration of the EX latent variables, and together, all comparisons reveal the latent variables themselves. Similarly, the relationships among the latent variables can be learned [2]. A more challenging problem is learning a dynamic (temporal) LVM, where the latent variables and their interrelations may change in time. A common graphical method to model dynamic

systems is the hidden Markov model (HMM) and its extensions, e.g., the factorial HMM, and the dynamic Bayesian network (DBN) [3] that generalizes the HMM model family. An advantage of a DBN over an HMM is its ability to represent multi variable complex models in a relatively easy manner. However, like the (static) search and score algorithms of learning an LVM (e.g., SEM), the learned DBN model may fit the data reasonably well, but miss representing causal relations. Thus, in this paper, we combine learning a static causal LVM and that of a DBN to learn a temporal LVM. More explicitly, to learn a temporal LVM, we learn a latent DBN by locally learning a MIM structure based on the LPCC algorithm and scoring the structure edges in each time slice, and then learning the parameters using a backward/forward algorithm.

2 Combining local to global graphs

Local to global learning (LGL) of a Bayesian network (BN) was introduced by [1] who suggested to learn locally (and separately) the Markov blanket (MB) of each variable in the BN and combine all local graphs/structures into the global graph. However, simple LGL approaches based on edge scoring are applicable only to fully observed data. For example, in the presence of latent variables, learning an LGL will lead to the wrong MBs (since a latent variable d-connects observed variables for which it is a common cause). Also, edge scoring is not applicable since the number of latent variables and their identities may change across local graphs. Therefore, we introduce a new method to merge local graphs into a global BN in the presence of latent variables. The local LVMs will be learned based on sequential time slices, and their unification will provide the (global) temporal LVM. Our proposed method makes three assumptions: 1) Data are assumed to be temporal and stationary 2) The model is a DBN, i.e., a pair (BN_1, $BN_{\rightarrow}$), where BN_1 defines the BN in $t = 1$ and $BN_{\rightarrow}$ is a two-slice temporal BN (2-TBN), with latent variables, and 3) Each slice in the 2-TBN is a pure measurement model (PMM).

Temporal and stationary data are appropriate for LGL, where a local graph is learned for each time slice and the global graph is a DBN. More explicitly, we suggest that each local graph would be a 2-TBN of two consecutive time slices because in DBN we need to learn both inter (between-slice) and intra (within-slice) edges. Since we base this local learning on the LPCC algorithm, we call our algorithm LGL-based LPCC. The LGL-based LPCC algorithm has two stages: in the first stage, it learns sets of observed that are children of the same latent variables (we call it MSO following [2]) while in the second, it finds the relationships between latent variables based on co-occurrences of these relationships across the local graphs. We demonstrate the algorithm using a synthetic example. Consider a dataset representing a temporal model with five slices ($T = 5$) and three observed variables per slice. Fig. 1 shows the results of learning four local graphs based on LPCC over $(t \cap t+1)$, $(t+1 \cap t+2)$, $(t+2 \cap t+3)$, and $(t+3 \cap t+4)$. The LGL-based LPCC algorithm merges these local graphs into a single 2-TBN by forming:

1. Matrix S (Table 1a) counts (across all local graphs) for each two observed variables the number of times they share a common latent variable (e.g., $X3$ and $X4$ share a common latent variable in two local graphs, as in Fig. 1(b) and 1(d), thus, $S(3,4) = S(4,3) = 2$).
2. A list LT per each observed variable is created to hold its most common observed variables in terms of sharing the same latent variable. The lists (due to Table 1a) are:
   1) $X1 \Rightarrow X2, X3$ 3) $X3 \Rightarrow X1, X2$ 5) $X5 \Rightarrow X6$
   2) $X2 \Rightarrow X1, X3$ 4) $X4 \Rightarrow X6$ 6) $X6 \Rightarrow X4, X5$.
3. LTs feed a final set, FS, of MSOs. $X1$, $X2$, and $X3$ (1) are entered first to FS. The next two lists (2 and 3) have no effect on FS. $X4$ and $X6$ are merged (4) before being merged

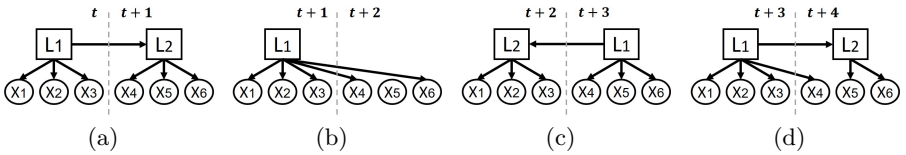


Fig. 1. Four local graphs for five slices and three observed variables per slice.

again with $X5$ (5) (chain rule). Note, that LTs are not necessarily symmetric (e.g., X_i can be in X_j's list, but not vice versa). Finally, FS is: $\{\{X1, X2, X3\}, \{X4, X5, X6\}\}$.

4. In the second stage, the co-occurrence matrix between latent variables is determined based on the local graphs. According to the first local graph (Fig. 1a), $L1 \rightarrow L2$, and thus the $(1, 2)$ entry in Table 1b increases in 1. Note that according to FS, $L1$ and $L2$ are associated with $\{X1, X2, X3\}$ and $\{X4, X5, X6\}\}$, respectively. The second local graph (Fig. 1b) adds no information about the relation between $L1$ and $L2$. The third local graph (Fig. 1c) also shows $L1 \rightarrow L2$, but since the connections of the observed variables to the latent variables are opposite in their associations according to FS, there is no support to $L1 \rightarrow L2$ but to $L2 \rightarrow L1$, and thus the $(2, 1)$ entry in Table 1b increases in 1. The fourth local graph (Fig. 1d) increases the $(1, 2)$ entry in Table 1b in 1 because three of the four connected observed variables, $\{X1, X2, X3\}$, provide stronger evidence in favor of $L1$ than the single connected observed variable, $X4$, provides in favor of $L2$. Because entry $(1, 2)$ is larger than entry $(2, 1)$, the algorithm learns $L1 \rightarrow L2$ in the final graph (in case of a tie, the edge remains undirected). Note that if the first and third local graphs were examined in the opposite order, we would have obtained the symmetric matrix to Table 1b but learned the same resulted graph ($''L1''$ and $''L2''$ are arbitrary indicated by LPCC). Thus, in this example, the final 2-TBN that LGL-based LPCC learns is the one that can be seen in Fig. 1(a).

Table 1. (a) Matrix S counts commonly shared latent variables for pairs of observed variables. (b) Matrix (CO) holding the co-occurrences counts of edges between latent variables.

	X1	**X2**	**X3**	**X4**	**X5**	**X6**
X1	-	4	4	2	0	1
X2	4	-	4	2	0	1
X3	4	4	-	2	0	1
X4	2	2	2	-	2	3
X5	0	0	0	2	-	3
X6	1	1	1	3	3	-

(a)

	L1	**L2**
L1	-	2
L2	1	-

(b)

References

1. C. F. Aliferis, I. Tsamardinos, S. Mani, and X. D. Koutsoukos. Local causal and Markov blanket induction for causal discovery and feature selection for classification. JMLR **1**, 235–284 (2010).
2. N. Asbeh and B. Lerner. Learning latent variable models by pairwise cluster comparison: Part I-theory and overview. JMLR **17**(224), 1–52 (2016).
3. K. P. Murphy. Machine learning a probabilistic perspective. MIT press, 2014.
4. R. Silva, C. Glymour, and P. Spirtes. Learning the structure of linear latent variable models. JMLR **9**, 191–246 (2006).

[8] Microsoft Corporation. 2017. Trojan:Win32/Kovter. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Kovter. (2017).
[9] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web*. ACM, ., 281–290.
[10] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection.. In *USENIX Security Symposium*. USENIX Association, ., 33–48.
[11] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, ., 3422–3426.
[12] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990), 179–211.
[13] Kunihiko Fukushima and Sei Miyake. 1982. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*. Springer, ., 267–285.
[14] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press, . http://www.deeplearningbook.org/
[15] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).
[16] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. 2005. Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition. In *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, 15th International Conference, Warsaw, Poland, September 11-15, 2005, Proceedings, Part II (Lecture Notes in Computer Science)*, Wlodzislaw Duch, Janusz Kacprzyk, Erkki Oja, and Slawomir Zadrozny (Eds.), Vol. 3697. Springer, ., 799–804. https://doi.org/10.1007/11550907_126
[17] Alex Graves and Navdeep Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. JMLR.org, ., 1764–1772.
[18] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, ., 6645–6649.
[19] Douglas M Hawkins. 2004. The problem of overfitting. *Journal of chemical information and computer sciences* 44, 1 (2004), 1–12.
[20] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).
[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[22] Michael Hopkins and Ali Dehghantanha. 2015. Exploit kits: the production line of the cybercrime economy?. In *Information Security and Cyber Forensics (InfoSec), 2015 Second International Conference on*. IEEE, ., 23–27.
[23] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).
[24] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. IEEE, ., 1725–1732.
[25] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 15.
[26] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent Convolutional Neural Networks for Text Classification.. In *AAAI*, Vol. 333. AAAI Press, ., 2267–2273.
[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[28] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.
[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[30] Peter Likarish, Eunjin Jung, and Insoon Jo. 2009. Obfuscated malicious javascript detection using classification techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*. IEEE, ., 47–54.
[31] Bing Liu and Lei Zhang. 2012. A survey of opinion mining and sentiment analysis. In *Mining text data*. Springer, ., 415–463.
[32] Christopher D Manning, Hinrich Schütze, et al. 1999. *Foundations of statistical natural language processing*. Vol. 999. MIT Press, .
[33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[34] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. ISCA, ., 3.
[35] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. NIPS, ., 3111–3119.
[36] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. Omnipress, ., 807–814.
[37] PaloAlto. 2017. Pulling Back the Curtains on EncodedCommand PowerShell Attacks. https://researchcenter.paloaltonetworks.com/2017/03/unit42-pulling-back-the-curtains-on-encodedcommand-powershell-attacks/. (2017).
[38] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. 2015. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, ., 1916–1920.
[39] Santiago M Pontiroli and F Roberto Martinez. 2015. The Tao of .NET and PowerShell Malware Analysis. In *Virus Bulletin Conference*. ., .
[40] Joseph D. Prusa and Taghi M. Khoshgoftaar. 2017. Deep Neural Network Architecture for Character-Level Learning on Short Text. In *Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2017, Marco Island, Florida, USA, May 22-24, 2017*. AAAI Press, ., 353–358.
[41] Amanda Rousseau. 2017. Hijacking. NET to Defend PowerShell. *arXiv preprint arXiv:1709.07508* (2017).
[42] Sam T Roweis and Lawrence K Saul. 2000. Nonlinear dimensionality reduction by locally linear embedding. *science* 290, 5500 (2000), 2323–2326.
[43] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[44] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*. ISCA, .
[45] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*. IEEE, ., 11–20.
[46] Joshua Saxe and Konstantin Berlin. 2017. eXpose: A Character-Level Convolutional Neural Network with Embeddings For Detecting Malicious URLs, File Paths and Registry Keys. *arXiv preprint arXiv:1702.08568* (2017).
[47] Robert J Schalkoff. 1997. *Artificial neural networks*. Vol. 1. McGraw-Hill New York, .
[48] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.
[49] Michael R Smith, Joe B Ingram, Christopher C Lamb, Timothy J Draelos, Justin E Doak, James B Aimone, and Conrad D James. 2017. Dynamic Analysis of Executables to Detect and Characterize Malware. *arXiv preprint arXiv:1711.03947* (2017).
[50] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. 2009. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, ., 635–647.
[51] Martin Sundermeyer, Tamer Alkhouli, Joern Wuebker, and Hermann Ney. 2014. Translation Modeling with Bidirectional Recurrent Neural Networks.. In *EMNLP*. ACL, ., 14–25.
[52] Symantec. 2016. The increased use of Powershell in attacks. https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf. (2016).
[53] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. 2016. A deep learning approach for detecting malicious JavaScript code. *Security and Communication Networks* 9, 11 (2016), 1520–1534.
[54] B Yegnanarayana. 2009. *Artificial neural networks*. PHI Learning Pvt. Ltd., .
[55] Xiang Zhang and Yann LeCun. 2015. Text understanding from scratch. *arXiv preprint arXiv:1502.01710* (2015).
[56] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*. NIPS, ., 649–657.

Why Are Repeated Auctions In RaaS Clouds Risky?

Danielle Movsowitz[1], Liran Funaro[2], Shunit Agmon[2], Orna Agmon Ben-Yehuda[1,2], and Orr Dunkelman[1]

[1] University of Haifa
dmovsowi@campus.haifa.ac.il, orrd@cs.haifa.ac.il
[2] Technion—Israel Institute of Technology
{funaro,shunita,ladypine}@cs.technion.ac.il

Extended Abstract

The Resource-as-a-Service (RaaS) cloud [8] is an economic model of cloud computing that allows providers to sell adjustable quantities of individual resources (such as CPU, RAM, and I/O resources) for short intervals—even at a sub-second granularity. In the RaaS cloud, clients can purchase exactly the resources they need when they need them. As price wars drive cloud providers towards this model [10], they start offering plans for dealing with resource requirement bursts: CloudSigma offered time-varying burst prices in 2010 [3], Amazon EC2 offered burstable performance instances in 2014 [4], Google Cloud offered Pay-as-you-go in 2016 [5], and Microsoft introduced the burstable Azure cloud Instance in 2017 [2]. When resources are dynamically rented, e-commerce requires calculating online economic decisions. Such decisions can only be made in real time by automated agents. E-commerce also requires efficient and computationally simple allocation mechanisms. These mechanisms may be centralized (as in an auction) or decentralized (as in a marketplace [16] or by negotiations [7]).

We see that horizontal scaling (adding more machines) has already matured to the point of incorporating advanced economic mechanisms such as auctions (e.g., AWS Spot Instances [9], Packet [6], and Alibaba Cloud Spot Instances [1]). Nevertheless, in the case of vertical scaling (increasing an existing machine's resources) we are only now seeing signs of early adoption of such mechanisms (e.g., Amazon EC2 T2 Instances, Google Cloud Platform, CloudSigma).

In the past few years, numerous studies have been published regarding different attack methods relevant to clouds, e.g., side channel [13], Resource Freeing Attack (RFA) [14], co-location attacks [15], and Economic Denial of Sustainability (EDoS) [12]. Most of the studied attacks are aimed at penetrating the security of the system and not at the economic mechanism that drives the resource allocation in the system. EDoS attacks are an exception: they cause victims to scale their resources beyond their economic means. In this work we take this line of vulnerabilities further, presenting combined economic-computer-science attacks.

Our contribution is the design of two low-cost economic attacks aimed at auction based clouds. The implementation and evaluation of the attacks were

done using Ginseng [11], a market driven cloud system for efficient RAM allocation. The first attack is the Price Raising attack. This attack raises prices in the system, thus reducing the victim's profit and forcing it to free resources. This enables the attacker to rent the freed resources at a negligible cost. The second attack is the Elbowing attack. This attack hinders the victim's performance by outbidding it for a single round at specific points in time. Due to the nature of RAM usage, the victim suffers from reduced performance even after the attack round ends, and it re-acquires the RAM. We demonstrate how the *Price Raising attack* reduces the victim's profit sevenfold and the *Elbowing attack* causes damage of $\$290 - \$630$ for every dollar spent on the attack.

The Elbowing attack can be applied to various economic mechanisms. Future works will try to amplify the effects of the Elbowing attack, e.g by coupling it with an additional attack which will inform the attacker of the optimal attack times. An optimal time for an attack like this depends on the quality and quantity of the evicted data. The RAM utilization is of high quality when the victim values its RAM usage the most. This valuation might be deduced from its bid price, or from side channels such as the victim's traffic volume or destination.

Acknowledgments

This work was partially funded by the Amnon Pazi memorial research foundation. We thank A. Schuster, D. Parkes and E. Tromer for fruitful discussions.

References

1. Alibaba cloud spot instances, `https://www.alibabacloud.com/help/doc-detail/52088.htm`, Accessed on 11.03.2018
2. Azure, `https://tinyurl.com/burstable-azure-cloud-instance`, Accessed on 03.06.2018
3. Charting CloudSigma burst prices, `https://kkovacs.eu/cloudsigma-burst-price-chart`, Accessed on 21.04.2018
4. EC2 instances with burstable performance, `https://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/`, Accessed on 11.03.2018
5. Google cloud platform, `https://cloud.googleblog.com/2016/09/introducing-Google-Cloud.html`, Accessed on 11.03.2018
6. Spot marketing pricing—discount packet bare metal servers., `https://www.packet.net/bare-metal/deploy/spot/`, Accessed on 02.06.2018
7. Agmon, S., Agmon Ben-Yehuda, O., Schuster, A.: Preventing collusion in cloud computing auctions. Tech. Rep. CS-2018-01, Technion (2018), `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2018/CS/CS-2018-01`
8. Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., Tsafrir, D.: The Resource-as-a-Service (RaaS) cloud. In: USENIX Conference on Hot Topics in Cloud Computing (HotCloud) (2012)
9. Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., Tsafrir, D.: Deconstructing Amazon EC2 spot instance pricing. ACM Transactions on Economics and Computation **1**(3), 16 (2013)

10. Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., Tsafrir, D.: The rise of RaaS: The Resource-as-a-Service cloud. Commun. ACM **57**(7), 76–84 (Jul 2014). https://doi.org/10.1145/2627422, http://doi.acm.org/10.1145/2627422
11. Agmon Ben-Yehuda, O., Posener, E., Ben-Yehuda, M., Schuster, A., Mu'alem, A.: Ginseng: Market-driven memory allocation. ACM SIGPLAN Notices **49**(7), 41–52 (2014)
12. Hoff, C.: Cloud computing security: From DDoS (distributed denial of service) to EDoS (economic denial of sustainability), https://tinyurl.com/from-ddos-to-edos, Accesses on 27.05.18
13. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 199–212. ACM (2009)
14. Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., Swift, M.M.: Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 281–292. ACM (2012)
15. Varadarajan, V., Zhang, Y., Ristenpart, T., Swift, M.M.: A placement vulnerability study in multi-tenant public clouds. In: USENIX Security Symposium. pp. 913–928 (2015)
16. Yu, D., Mai, L., Arianfar, S., Fonseca, R., Krieger, O., Oran, D.: Towards a network marketplace in a cloud. In: HotCloud (2016)

Preserving Differential Privacy and Utility of Non-Stationary Data Streams

Michael Khavkin[1] and Mark Last[2]

[1] Ben-Gurion University of the Negev, Be'er-Sheva 85104, Israel
khavkin@post.bgu.ac.il
[2] Ben-Gurion University of the Negev, Be'er-Sheva 85104, Israel
mlast@bgu.ac.il

Abstract. Data publishing poses many challenges regarding the efforts to preserve data privacy, on one hand, and maintain its high utility, on the other hand. The Privacy Preserving Data Publishing field (PPDP) has emerged as a possible solution to such trade-off, allowing data miners to analyze the published data, while providing a sufficient degree of privacy. Most existing anonymization platforms deal with static and stationary data, which can be scanned at least once before its publishing. More and more real-world applications generate streams of data which can be non-stationary, i.e., subject to a concept drift. In this paper, we introduce MiDiPSA (Microaggregation-based Differential Private Stream Anonymization) algorithm for non-stationary data streams, which aims at satisfying the constraints of k–anonymity, recursive (c, l)–diversity, and differential privacy while minimizing the information loss and the possible disclosure risk. In our empirical evaluation, we analyze the performance of various data stream classifiers on the anonymized data and compare it to their performance on the original data. We conduct experiments with seven benchmark data streams and show that our algorithm preserves privacy while providing higher utility, in comparison with other state-of-the-art anonymization algorithms.

Keywords: Privacy-Preserving Data Publishing · Concept Drift · Differential Privacy · Microaggregation · k–Anonymity · (c, l)–Diversity

1 Introduction

Publishing collected data is not trivial anymore, since it can pose some risks to individual's privacy, such as sensitive information leakage or possible inference by adversaries. Moreover, simply removing any explicit identifiers, such as name or ID, is not enough to guarantee user's privacy.

The PPDM (Privacy-Preserving Data Mining) field [1] concentrates on masking the output of any interactive mining mechanism, providing the data miner with an anonymized version of the results. A related extension to PPDM, PPDP (Privacy-Preserving Data Publishing) [6] has been introduced as a more flexible non-interactive mechanism, in which data is anonymized and later published for further research or analysis, without depending on a specific data mining task.

In this paper, we present MiDiPSA (Microaggregation-based Differential Private Stream Anonymization) for publishing non-stationary [7] data streams, balancing between privacy and utility. Post-analysis of the anonymized tuples is performed using four stream classifiers which are trained on the published data and tested relatively to their performance on the original data.

2 Microaggregation-based Differential Private Stream Anonymization (MiDiPSA) for non-stationary data

Based on [4], our algorithm [3] comprises four steps: In the first step, incremental clustering of the tuples in the stream is performed, searching for the closest cluster that minimizes the incurred anonymization information loss. In the second step, the tuples in each cluster are aggregated to a centroid, representing the cluster. Then, statistical multivariate test is used for detecting concept drift in each cluster. Finally, noise from Laplace mechanism [5] is added to the centroid of each cluster, satisfying k–anonymity [11] and (c, l)–diversity [9], before publishing the tuples in the cluster.

3 Evaluation

We evaluate the performance of the proposed anonymization algorithm on four real and three synthetic stream datasets, previously utilized by the Massive Online Analysis software (MOA) [2]. Each dataset was used to simulate a stream, handling one tuple at-a-time. Information loss and the disclosure risk of the published anonymized tuples were measured for assessing privacy vs. utility. Average publishing delay of the tuples was also measured, as an indication of the feasibility of our method in real-world applications that often require minimum delay.

Fig. 1(a) compares the incurred information loss of the published tuples, relatively to their original instances. While the results of CASTLE, BCASTLE and FADS show an increase in the information loss for higher values of k, the KNN-based algorithm and MiDiPSA show an opposite trend. The decreasing trend relates to the use of random noise, added to each centroid before publishing, as a result of applying the differential privacy mechanism. The larger the cluster, the smaller the impact of noise on the cluster tuples and their centroid, representing a lower sensitivity of the release mechanism. Hence, our algorithm (MiDiPSA) shows the minimal information loss out of all evaluated algorithms.

Fig. 1(b) compares the average publishing delay of MiDiPSA and the four other algorithms. Although all algorithms behave similarly for small clusters (related to low values of the k–anonymity parameter), MiDiPSA achieves a shorter publishing delay for larger clusters (at $k = 800$). This can be attributed to the fact that we distinguish between small, large and exactly k–tuple clusters, incrementally publishing each tuple arriving at a k–tuple cluster.

[3] Available at https://github.com/MichaKh/MiDiPSA-for-non-stationary-streams

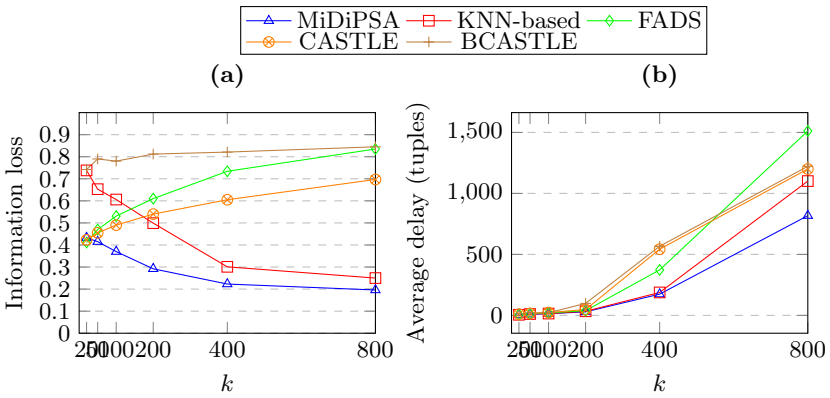


Fig. 1: (a) Information loss and (b) average publishing delay on "Airlines" stream for increasing values of k

A a post-analysis, each published stream was used to prequentially train four stream classifiers, using the MOA [2] software. These include Naive Bayes, Adaptive Random Forest, CVFDT (Concept-Adapting Very Fast Decision Tree), and the online Perceptron classifier.

Stream	**IL**		**DR**		**Utility**	
	MiDiPSA	KNN-based	MiDiPSA	KNN-based	MiDiPSA	KNN-based
Adult1	0.140	**0.088**	0.012	**0.010**	**0.608***	0.507
Adult2	**0.209***	0.566	0.244	**0.011***	**0.613***	0.564
Electricity	**0.097***	0.255	0.526	**0.029***	**0.638***	0.570
Airlines	**0.322***	0.508	0.254	**0.012***	**0.548***	0.433
SEA	**0.183**	0.234	0.136	**0.010**	**0.677***	0.531
Hyperplane	**0.193***	0.357	**0.007**	0.011	**0.512***	0.459
Random RBF	**0.060**	0.106	**0.005**	0.010	**0.695***	0.502

Table 1: Trade-off between information loss and disclosure risk and its impact on the AUC of Adaptive Random Forest classifier (* p –value < 0.05)

The anonymization performance was compared to four other anonymization algorithms: CASTLE [3], BCASTLE [12], the Fast clustering-based k–Anonymization approach (FADS) [8], and the recently developed KNN-based differentially-private microaggregation [10].

Table 1 describes the trade-off between information loss and disclosure risk, on all the seven datasets, and its impact on the Area Under the ROC Curve (AUC) of Adaptive Random Forest classifier, trained on both our and the KNN-based published tuples. Friedman statistical test has been performed to verify the significance of the results.

4 Conclusions

In this paper, we presented MiDiPSA (Microaggregation-based Differential Private Stream Anonymization) for continuously publishing non-stationary data. In addition to satisfying k–anonymity and (c, l)–diversity, we proposed a novel publishing technique, adhering to the conditions of $\epsilon-$differential-privacy, and an unsupervised mechanism for detection of concept drift under the evolving nature of the data. Our experiments evaluated the trade-off between privacy, measured by the disclosure risk, and utility, measured by the AUC of classifiers trained on the anonymized streams. The MiDiPSA algorithm outperformed other state-of-the-art anonymizers, achieving a lower information loss in most cases, but, on the other hand, a higher disclosure risk for some datasets. In addition, a shorter publishing delay was experienced using MiDiPSA for all datasets, a property that makes MiDiPSA adequate for real-world massive stream applications.

References

1. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: ACM Sigmod Record. vol. 29, pp. 439–450. ACM (2000)
2. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: Moa: Massive online analysis. Journal of Machine Learning Research **11**, 1601–1604 (2010)
3. Cao, J., Carminati, B., Ferrari, E., Tan, K.: Castle: Continuously anonymizing data streams. IEEE Transactions on Dependable and Secure Computing **8**(3), 337–352 (2011)
4. Domingo-Ferrer, J., Mateo-Sanz, J.M.: Practical data-oriented microaggregation for statistical disclosure control. IEEE Trans. on Knowl. and Data Eng. **14**(1), 189–201 (2002)
5. Dwork, C.: Differential privacy in new settings. In: Proceedings of the twenty-first annual ACM-SIAM Symposium on Discrete Algorithms. pp. 174–183. SIAM (2010)
6. Fung, B., Wang, K., Chen, R., Yu, P.S.: Privacy-preserving data publishing: A survey of recent developments. ACM Computing Surveys (CSUR) **42**(4), 14 (2010)
7. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Computing Surveys (CSUR) **46**(4), 44 (2014)
8. Guo, K., Zhang, Q.: Fast clustering-based anonymization approaches with time constraints for data streams. Knowledge-Based Systems **46**, 95–108 (2013)
9. Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: l-diversity: Privacy beyond k-anonymity. ACM Transactions on Knowledge Discovery from Data (TKDD) **1**(1), 3 (2007)
10. Rodríguez, D.M., Nin, J., Nuñez-del Prado, M.: Towards the adaptation of sdc methods to stream mining. Computers & Security **70**, 702–722 (2017)
11. Sweeney, L.: k-anonymity: A model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems **10**(05), 557–570 (2002)
12. Wang, P., Lu, J., Zhao, L., Yang, J.: B-castle: An efficient publishing algorithm for k-anonymizing data streams. In: Second WRI Global Congress on Intelligent Systems. vol. 2, pp. 132–136. IEEE (2010)

Using Deep Reinforcement Learning to Train Classifiers

Anton Puzanov, Kobi Cohen

Electrical and Computer Engineering, Ben Gurion University, Beer-Sheva, Israel
{poznov,yacovsec}@post.bgu.ac.il

Abstract. Human level performance in a variety of tasks has been demonstrated by computerized systems. Those advancements have been achieved due to significant improvement in computation power and increasing focus of the research community over the past two decades. One of the most prominent approach for online learning is Reinforcement Learning (RL), in which an agent learns good strategies by interacting with its environment by taking a series of rewarding actions. This approach works well when an optimal behavior is required but the problem domain cannot be modeled effectively, e.g. since the dimension is too high, or that states are partially observable. RL-based classifiers make it possible to consider different constrains in the training phase, but as it turns out, those models are highly affected by the environment parameters, obligating the designer to take extra care when modeling it. This paper presents possible design flaws and suggests best practices for training such models.

Keywords: Reinforcement learning, classification, active learning, one-shot learning

1 Introduction

Recent advancements in Artificial Intelligence (AI) and Machine Learning (ML) made it possible for computerized systems to perform many tasks at human level. Applications span from autonomous vehicles to chat bots. The extensive research of ML introduced a variety of new algorithms and frameworks. Reinforcement Learning (RL) is one of the most prominent among them which does not rely on concrete modeling of the environment. In RL, the trained agent interacts with the environment by means of actions and rewards, allowing the agent to learn an optimal policy even when its environment is not completely modeled. RL agents were able to play video games from raw pixels [1] and perform other complicated tasks. Recently, RL agents were suggested to act as a classifier with strict constrains in [2], where the authors successfully trained an agent to perform active one-shot learning. The one-shot learning problem deals with learning a class from a relatively small amount of labeled samples, which results in a complex task that usually employs a memory component [3]. Active Learning (AL) handles the situation where not all the samples have labels and the system must choose the samples which are labeled. In the online setting of active learning, the agent must choose whether to label a sample or request the true label [2]. The RL-classifier approach in [2] incorporates the label selection component inside the

This research was supported by the Cyber Security Research Center at Ben Gurion University of the Negev.

agent and automatically models the environment. However, as it turns out, this model is highly affected by the environment parameters, obligating the designer to take extra care when modeling it. This paper presents possible design flaws and suggests best practices for training such models.

2 Related Work

We adopt a model-free RL approach in which observations are mapped directly to actions. The observation-action mapping is commonly learned by a Q-Learning method where an estimated reward function, i.e. the Q function, is learned from the samples [4]. DQN is used to model the Q-function by incorporating Neural Networks (NN) that allows model generalization and space efficiency. Recently, RL agents were used to perform constrained classification tasks [2], by constraining the number of labeled samples that forces the agent to choose the most informative ones. Modern techniques consider uncertainty, diversity, and density of the samples to be labeled [5]. One-shot learning algorithms usually employ meta-learning with memory components [2,3] in which some features of past samples are stored for future comparison. The model suggested in [2] uses LSTM as the memory component, which makes it possible to learn long term dependencies between actions. The cells consisting of memory units, input, output, and forget gates, used for updates and interactions with other cells.

3 Preliminaries

We start by providing a short background on Q-learning. For a set of possible states S and allowed actions A, RL consists of experience tuples, given by $<s_t,a,r,s_{t+1}>$, where $s_t,s_{t+1} \in S$ are the current and next states, $a \in A$ is an action taken at time t which results in getting reward r. Q-Learning is used to predict the long-run reward when taking action at state *s*, which is called the Q-function. The prediction is given by $Q^*(s,a)$. Obtaining the policy is done by choosing the highest rewarding action and is defined by $V^*(s)=\text{argmax}_a Q^*(s,a)$. Updating the Q-function is done by minimizing the time-difference error

$$Q(s,a):=Q(s,a)+\alpha(r+\gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})-Q(s,a)) \quad (1)$$

, where α is the learning rate and γ is the discount factor.

4 Experiments and Results

The experiments conducted in this research rely on the model suggested in [2], and was evaluated with the Omniglot database. This databse consists of letters taken from 30 languages, where every letter has only 20 samples. An experiment consists of 30 images

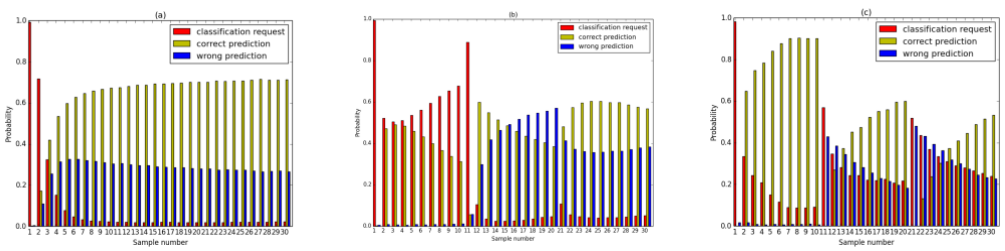


Fig. 1. Probabilities for classification request, correct prediction and wrong prediction, for every sample in the experiment; (a) shuffled examples from 3 classes; (b) ordered examples from 3 classes; (c) ordered examples from 3 classes out of 30.

from 3 classes arriving in a sequence. The samples may be shuffled or ordered, and the labels are randomly chosen. The agent may either try to predict the label or request it, and every evaluated sample consists of a concatenation between the image and a vector with a size equals to the number of classes. The tuple (x_t,y_t) represents the image and the label at time t, respectively. When the agent requests the label, the concatenated vector at time $t+1$ turns into one-hot vector which corresponds to y_t. The agent obtains rewards of $\{1,-1,-0.05\}$ upon correct prediction, wrong prediction, and classification request, respectively. The agent was trained for 300,000 epochs, learning rate of 0.001, with Adam optimizer, Fig 1.a shows the results for shuffled episodes, where the classifier requests less labels as it sees more samples. Fig 1.b shows the same experiment with ordered samples. Interestingly, at sample 21, the classifier is asked to label a previously unseen example, and yet it has 0.5 probability to correctly label the sample and almost never requests a label. This is one of the differences between agents and classifiers, where the learned policy understands that the new sample is different from the previously seen examples and since there must be 3 classes it guesses the label correctly. The probability of requesting a label in the first class increases as more examples have passed, meaning that the agent has learned the probabilities of the dynamics. Since the training used shuffled samples, the probability of seeing similar examples in all first requests is small and the agent concludes that the new examples are sampled from relatively similar but different classes. This results in requesting the label more often. We point out that designing a random environment is essential for preventing the over-fitting effect. Fig 1.c shows the performance of our improved agent, where the class labels were randomly chosen from 30 possibilities rather than 3, and when in the shuffled training phase, one of the classes was added to the experiments after that a random number of samples were already classified. This agent performed similarly under the shuffled experiment but did not guess the label on the 21st sample. It was able to correctly predict the label even when only a single image passed the system.

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. Playing atari with deep reinforcement learning. In Proc. of the Conference on Neural Information Processing Systems, workshop on Deep Learning, 2013.
2. Woodward, M., Finn, C. Active One-shot Learning. In Proc. of the Conference on Neural Information Processing Systems, workshop on Deep Learning, 2016.
3. Vinyals, O., Blundrell, C., Lillicrap, T., Kavukcuoglu, K., Wierstra, D., Matching Networks for One-Shot Learning. In Proc. of the Conference on Neural Information Processing Systems, 2016.
4. Kaelbling, P. L., Littman, L. M., Moore, W. A. Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research 4:237-285, 1996.
5. Demir, B., Bruzzone, L., A Novel Active Learning Method in Relevance Feedback for Content-Based Remote Sensing Image Retrieval, IEEE Transactions on Geoscience and Remote Sensing, vol. 53, no 5, 2323-2334, 2015.

Entrepreneurship Pitch Track

Chair: Sitaram Chamarty and Ben Gilad

Introduction

Entrepreneurship Pitch Track chaired by Ben Gilad and Sitaram Chamarty

Information security practitioners have always had to take a wide-angle view of the world, because there is no aspect of life that is so disconnected from information security that it is immune to problems, or at least the possibility of improvements.

This year's CSCML Entrepreneurship Pitch Track is a perfect example of this. It is clear that the entrants had a pulse on the current situation in information security, and are gearing up, (in some cases already have geared up) to meet the challenge head-on, and in the process protect all of us.

Entries ranged from computer vision to scheduling and planning, from SCADA and similar systems to online social media. It was heartening to note that, even in a business focused track, there were two entries that could justifiably be considered "for the greater good of the people" – that is, even if they had business motives and priorities, they would still end up benefiting all of us.

These entrepreneurs deserve all the encouragement that we in the community can give them, in whatever form is suitable.

As was the case last year, the Entrepreneurship Pitch Track at CSCML 2018 did an excellent job of fulfilling this objective, and consequently was a great success. It received sponsorship from leading VCs (JVP, BaseCamp Innovation Center, Telekom Capital Partners,) and corporations (DELLEMC, IBM, Tata Consultancy Services). Five start-ups pitched in the event, out of which 'HC Vision' was selected by the Entrepreneurship Pitch Track Committee as the leading entry. 'HC Vision' pitched in front of the entire audience who voted for the first (and second) best startup. 'HC Vision' presented readiness to turn their technological solution into a business and emerged as the winner! All three leading projects received a certificate of recognition.

We look forward to an even better CSCML 2019.

Regards,

Ben Gilad, Sitaram Chamarty
Tata Consultancy Services
Entrepreneurship Pitch Track Chairs

MATERIAL SENSING CAMERA

CSCML 2018 Entrepreneurship Pitch

HC VISION

HC VISION

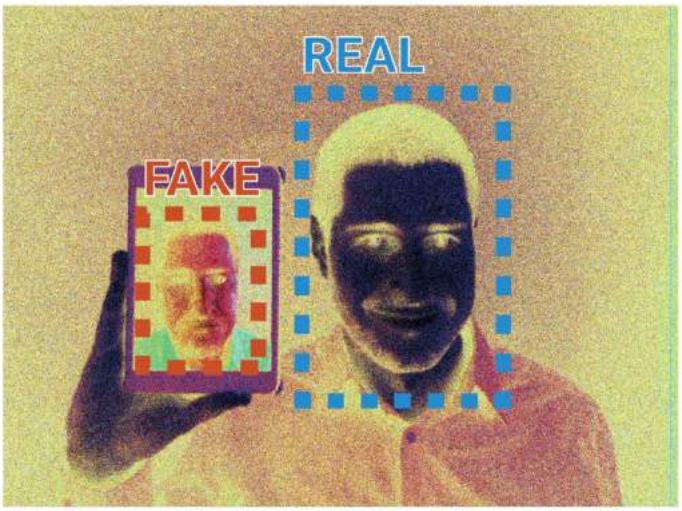
Current cameras still struggle to capture **dark scenes** and state of the art computer vision technology can only provide **limited information** based on the **shape and color** of objects.

Our **image processing software** that can dramatically improve **light sensitivity** and provide **material sensing capabilities** to existing cameras.

HC VISION

Secure Identification

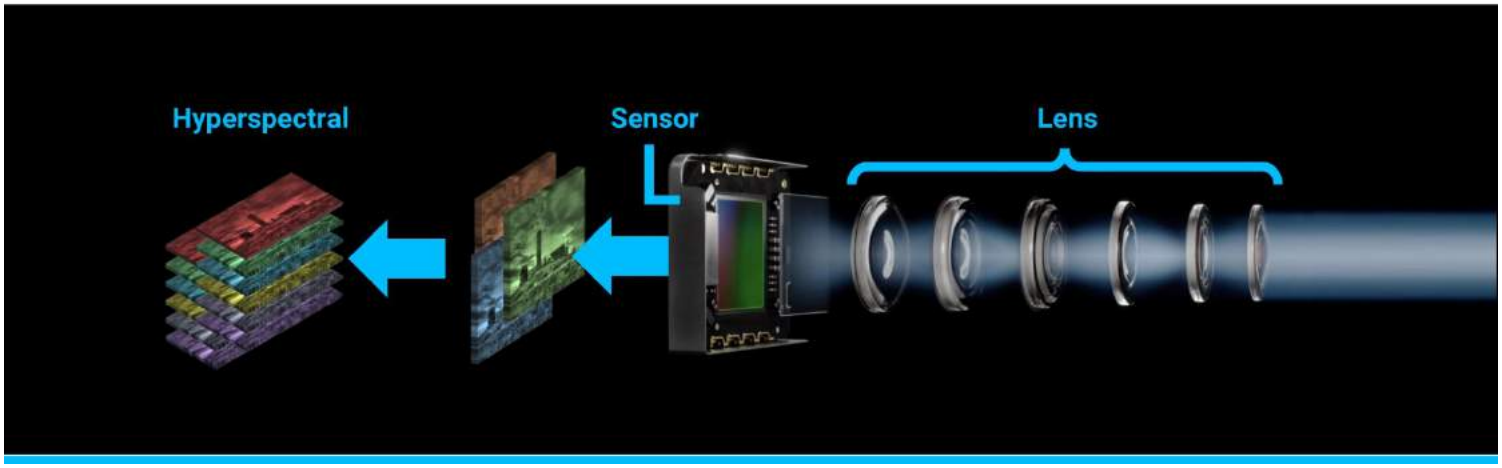
Distinguish between real and "fake" faces



Automotive Applications

Material information
distinguishes between similar objects

Patented compressed sensing technology developed here at Ben-Gurion university – allows the recovery of **hyperspectral** information from low-cost images sensors.



HC VISION

Thank you!

Ohad Ben-Shahar
Professor
Dept. of Computer Science,
Ben-Gurion University

Boaz Arad
M.Sc.
Dept. of Computer Science,
Ben-Gurion University

Omer Shwartz
M.Sc.
Dept. of Software Information Systems Engineering,
Ben-Gurion University

TOP INNOVATION 2018
Garage+
SILICONNEGEV
כלכליסט
גלובס

Distributed Dynamic Dispatch

Team

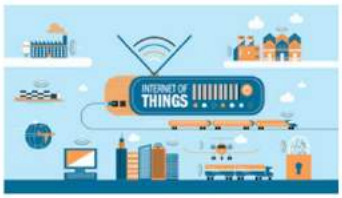
Dr. Roie Zivan

PhD. in Computer Science, BGU, 2007
Post-Doc – Robotics Institute, CMU, 2010.
Senior Lecturer at IE&M dep., BGU
Expertise: Distributed Artificial Intelligence and Multi Agent Systems.

Dr. Sofia Amador Nekle

PhD. in IE&M, BGU, 2017
Lecturer at IE&M dep., HIT
Expertise: Game Theory and Market Based Task Allocation.

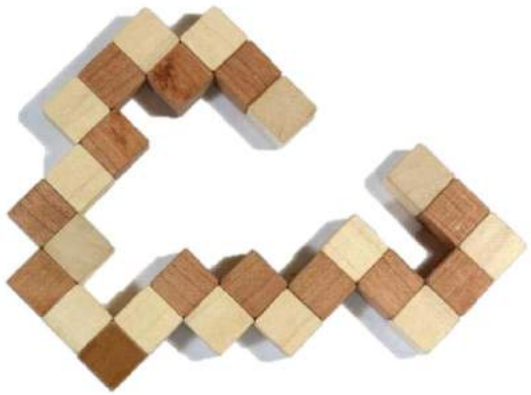
Relevant Applications



- Home Land Security:
  - Police
  - Fire Fighters
  - Anti Terror
  - Disaster Response
- Smart Ports
- Smart Factories
- Any Heterogeneous Service Providing system Requiring:
  - Ad-hoc heterogeneous service teams.
  - Handdling Temporal and Spatial Constraints

The Need

- Task allocation to teams
  (or autonomous devices)
- Heterogeneous teams
  Managing heterogeneous teams
- Upcoming tasks
  Handling upcoming tasks / developments in current tasks
- Collaboration
  Effective collaborative execution
- Avoiding Single point of failure
  e.g., in case of operations room collapses



The Solution

- Dynamic Distributed Task Allocation System

- Real time immediate assignment
- Effective utilization of resources
- Heterogeneous resources
- Centralized or distributed implementation (similar results).
- Can be applied in: Vehicles, Robots, Wearable devices, etc.

Current Status

- POC - Beer -Sheba police scenario.
- Patent Pending (PCT)
- Two scientific papers published in leading AI conferences.

Competitors -Homeland Security

	Heterogeneous Agents	UI	Smart Dispatcher (DSS)
DDD	✓	✓	✓
NowForce		✓	
Hexagon	✓	✓	
ZOLL		✓	

Thank You!

For more details:
Dr. Roie Zivan , CTO
zivanr@bgu.ac.il
050-9579209

2018 International Symposium On Cyber Security Cryptography and Machine Learning (CSCML 2018)

CBG
Cyber@Ben-Gurion University
Telekom Innovation Labs

Fake News Measurement Using Topic Authenticity

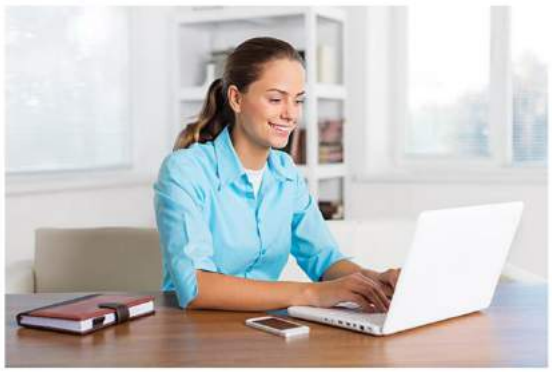
Aviad Elyashar, Jorge Bendahan, and Rami Puzis

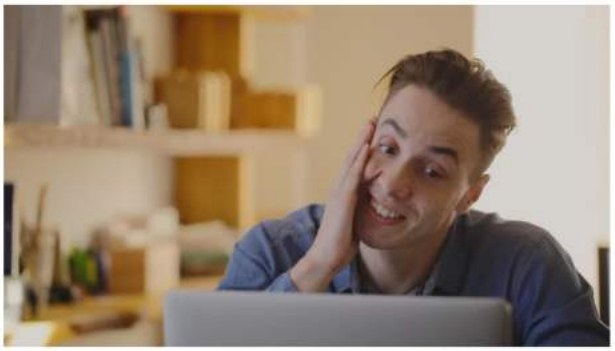
CSCML2018

SISE

THE PROBLEM

- Today, people tend to consume news from social media, rather than traditional news.





THE PROBLEM

- Today, people tend to consume news from social media, rather than traditional news.
- The nature of online news publication has changed, to the point that traditional fact checking and vetting are sometimes incomplete due to the flood of material from content aggregators

THE PROBLEM

- Therefore, fake news are widely spreading.
- Fake news detection on social media has recently become an emerging research that is attracting tremendous attention.

FAKE NEWS DEFINITION

- Particular news articles, which are intentionally written false and could mislead readers to believe false information (Allcot & Gentzkov, 2017).
- Satire news as fake news (Rubin et al. 2016)
- Deceptive news as fake news, which includes serious fabrications, hoaxes and satires (Rubin et al., 2015).

DANGERS ASSOCIATED WITH FAKE NEWS

- **Breaking the authenticity balance of the news ecosystem**

DANGERS ASSOCIATED WITH FAKE NEWS

- Breaking the authenticity balance of the news ecosystem
- **Reputational damage caused by spreading rumors.**

DANGERS ASSOCIATED WITH FAKE NEWS

- Breaking the authenticity balance of the news ecosystem
- Reputational damage caused by spreading rumors.
- **Fake perception**
  - **Fake news is usually manipulated to convey political messages or influence.**

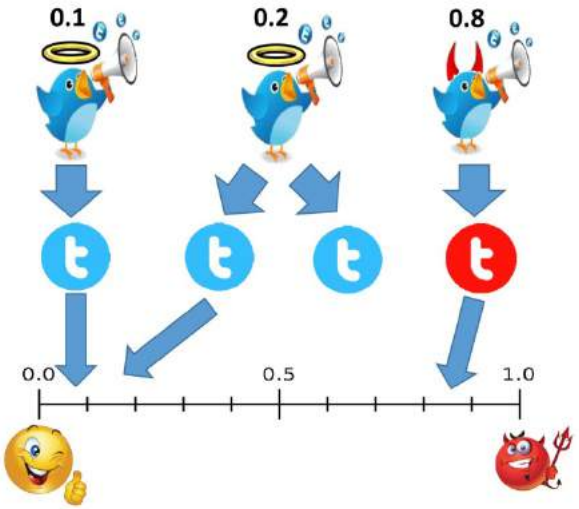
DANGERS ASSOCIATED WITH FAKE NEWS

- Breaking the authenticity balance of the news ecosystem
- Reputational damage caused by spreading rumors.
- **Fake perception**
  - Fake news is usually manipulated to convey political messages or influence.
  - **Spreading disinformation and propaganda**

TOPIC AUTHENTICITY

Measuring fake news in online social media based on estimating the distribution of fake news promoters among the accounts that contributed to the given online discussion.

TOPIC AUTHENTICITY



Aviad Elyashar, Jorge Bendahan, Rami Puzis, and Maria-Amparo Sanmateu, "**Measurement of Online Discussion Authenticity within Online Social Media**", 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM) 2017.

TOPIC AUTHENTICITY

Aviad Elyashar, Jorge Bendahan, Rami Puzis "**Is the News Deceptive? Fake News Detection Using Topic Authenticity**", SOTICS 2017.

QUESTIONS?

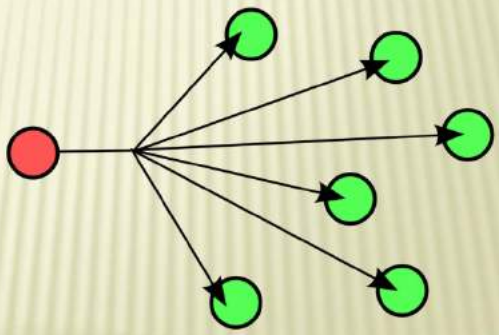
MY-BRDKSTV

Patent Inventors
Dan Brownstein
Shlomi Dolev
Niv Gilboa

Ben-Gurion University
of the Negev

FULL REVOCATION SYSTEMS

- Non-formal introduction: Model and different settings
- Contribution
- Related work
- Our Solution



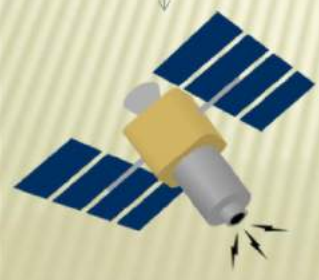
INTRODUCTION

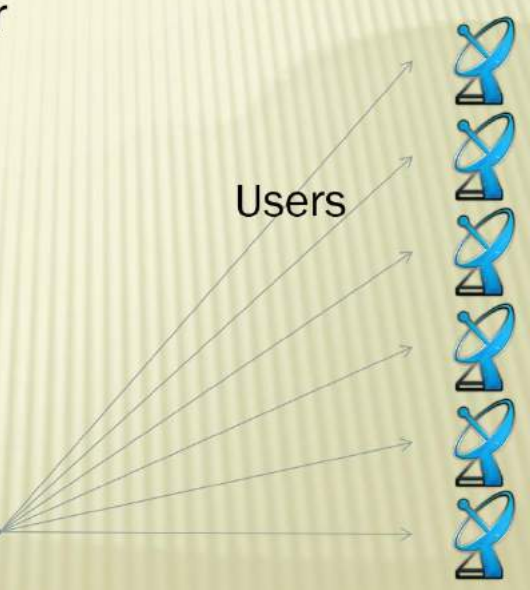
Content provider

FiatNaor93

Users

Broadcaster





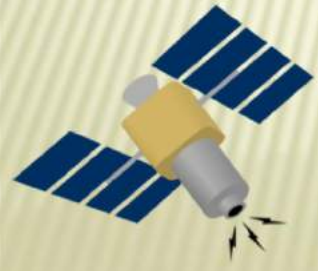
INTRODUCTION- EFFICIENCY

Encryption Key

Encryption Time

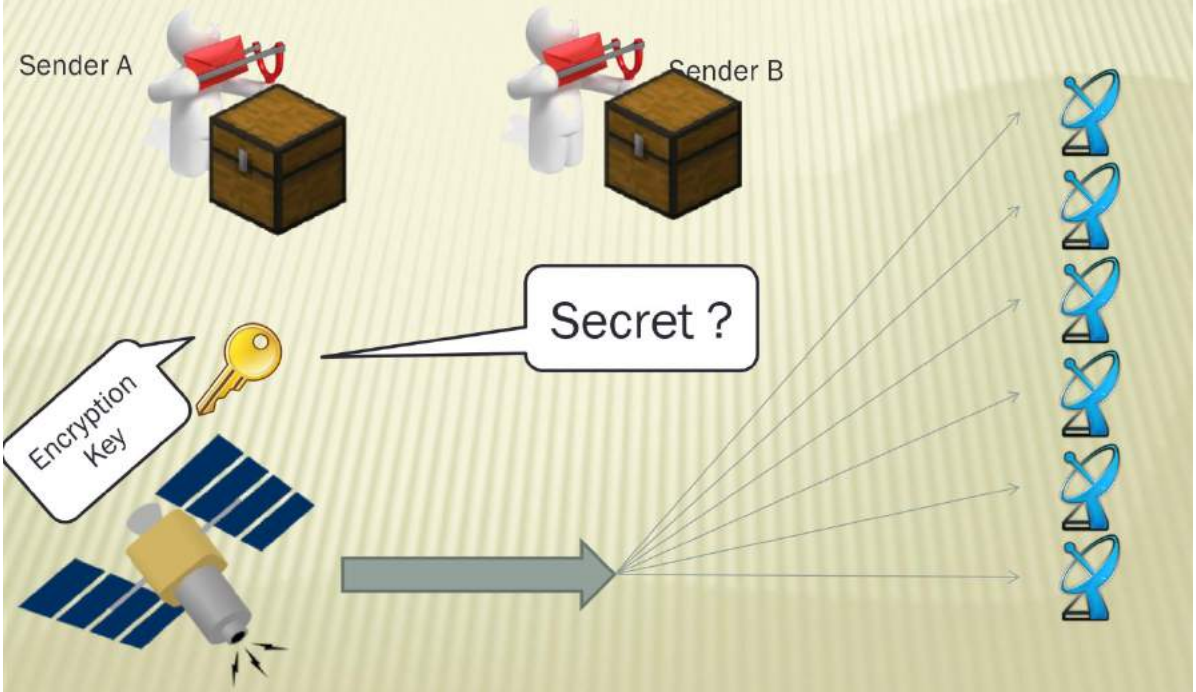
Decryption Time

Decryption Key



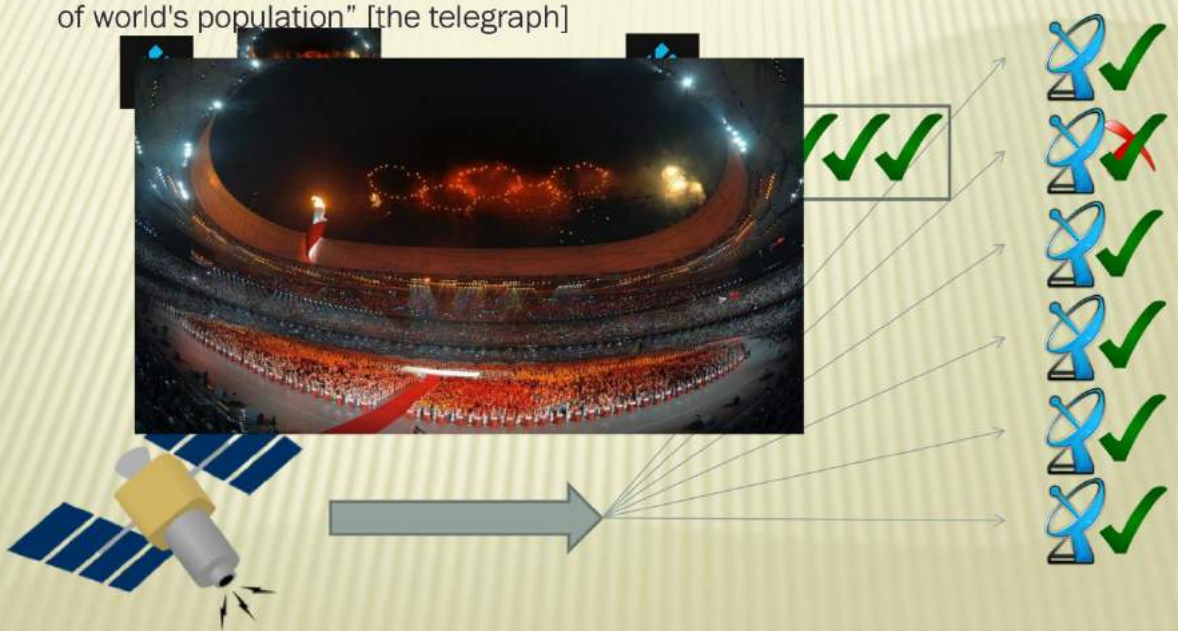
Ciphertext

INTRODUCTION- SYMMETRIC VS. PUBLIC-KEY

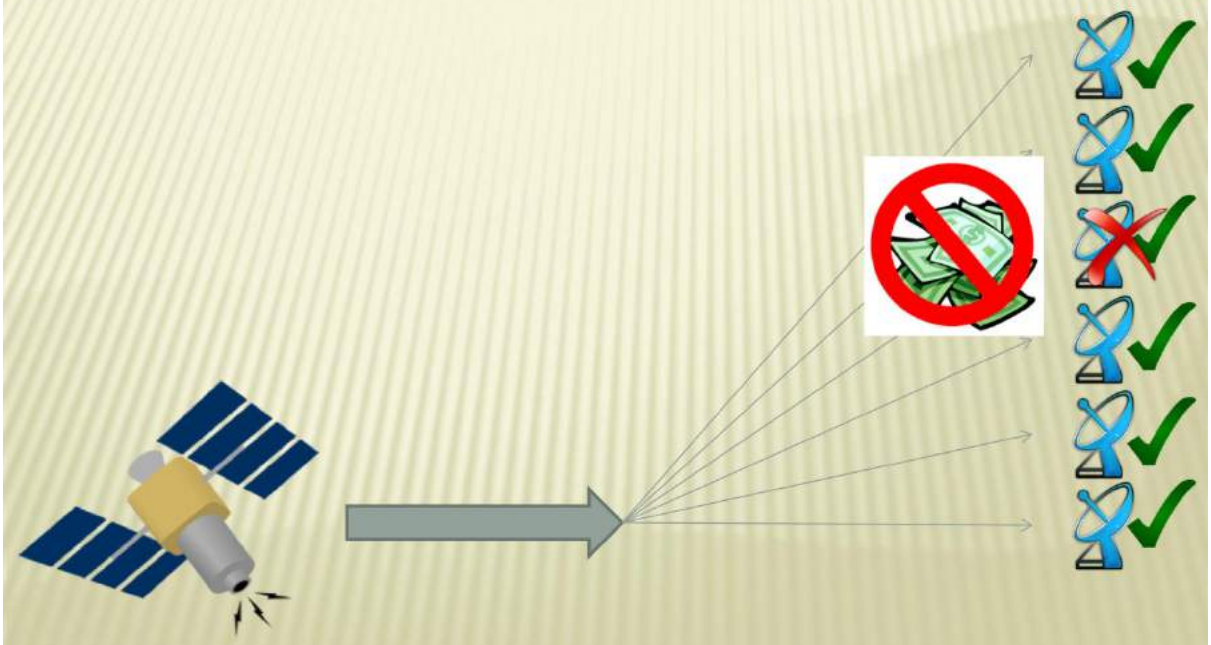


INTRODUCTION- TEMPORARY REVOCATION

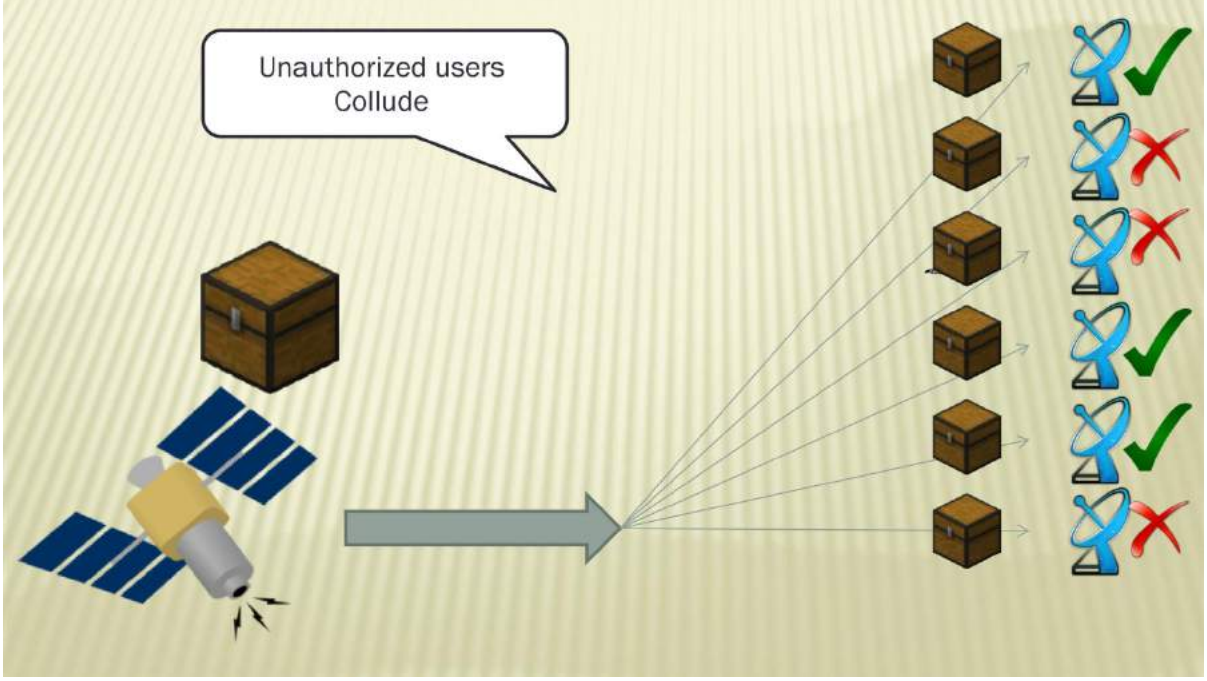
"Beijing Olympics: Opening ceremony watched by 15 percent of world's population" [the telegraph]



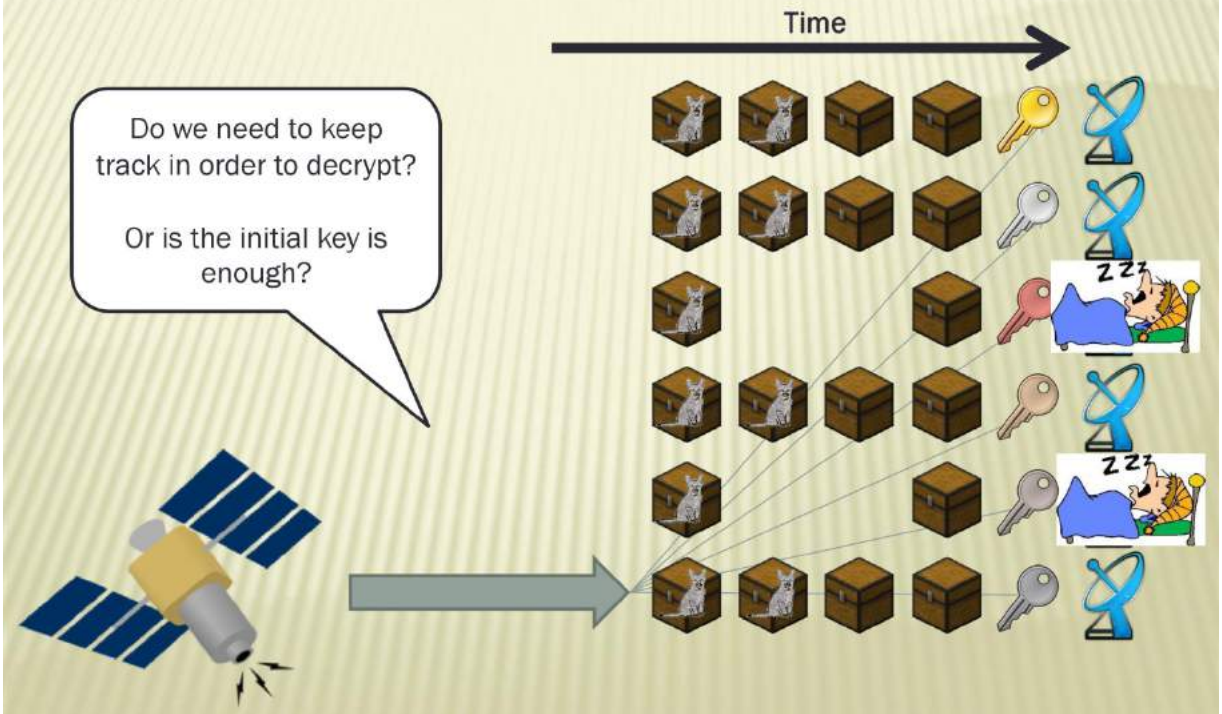
INTRODUCTION- PERMANENT REVOCATION



INTRODUCTION- COLLUSION RESISTANCE

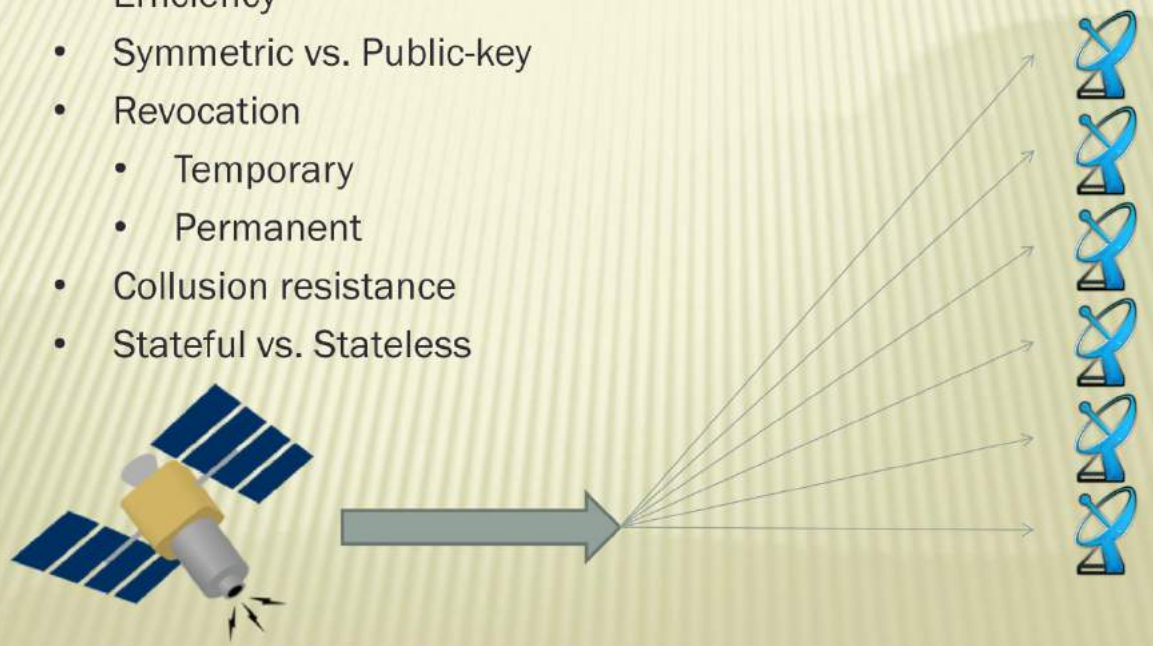


INTRODUCTION- STATEFUL VS. STATELESS



FULL REVOCATION SYSTEMS

- Efficiency
- Symmetric vs. Public-key
- Revocation
  - Temporary
  - Permanent
- Collusion resistance
- Stateful vs. Stateless



INDUSTRIAL AND MARKET OPPORTUNITIES

A platform for broadcasting live events such as sport matches and concerts.

Usually, content providers of such events make their profit (content, not merchandise wise) by selling tickets and recorded videos of the events.

With our scheme, they can also make profit by broadcasting the events in real-time and thus reach clients that are unable to attend the events (there are many reasons
for that i.e., sold out or too expensive tickets, geographic distance and more).

Furthermore, it might help individuals, who are unable to attend such events, to enjoy the experience of watching them live at reasonable prices. Another possibility is for content providers to sell the broadcast rights to hang-out places such as bars and restaurants who can then offer their clients a richer experience and attract more clients.

INDUSTRIAL AND MARKET OPPORTUNITIES

A short survey of the market size of live broadcast shows that the implications of our scheme can be critical . Some examples are: "The global video streaming software market size is expected to grow from USD 3.25 Billion in 2017 to USD 7.50 Billion by 2022, at a Compound Annual Growth Rate (CAGR) of 18.2%"

"Live streaming video will further accelerate streaming videos overall share of internet traffic. Streaming video accounts for over two-thirds of all internet traffic, and this share is expected to jump to 82% by 2020, according to Ciscos June 2016 Visual Networking Index report...

INDUSTRIAL AND MARKET OPPORTUNITIES

While live streaming is still in its early stages, brands are leveraging micropayments, mid-roll video ads and direct payments from social platforms, to monetize their live streaming videos"

.

"More broadcasters moving into live streaming (Over the Top) OTT video... We expect some big changes in the video industry, and specifically in live streaming this coming year...

Streaming Media has declared SVOD (Subscription Video on Demand) the best business model for OTT video

Detection of High-level Anomaly Events in Utility Networks

Michael Orlov

Shamoon College of Engineering, Beer Sheva, Israel
orlovm@noexec.org

Abstract. This entrepreneurship pitch submission describes a POC for detecting high-level anomaly events in utility networks such as power grids, water plants, pressurized pipelines, etc.

Keywords: Cyber anomalies · Machine learning · Power grids.

1 Introduction and Project Description

Utility networks with complex topologies, such as heterogeneous electricity grids, wide-area water pipelines, and so forth, are vulnerable to cybersecurity intrusions [3]. A defining feature of such intrusions is that while simple intrusions are fairly easy to detect, since they result in local spikes that are inconsistent with the global grid configuration, extensive intrusions are much harder to detect. Local spikes exist in such attacks, but stand out less, and are coordinated in order to prevent automatic detection.

The proposed solution is to differentiate between detection of unreliable low-level events, and detection of high-level intrusions at a higher abstraction level that is created by automatically learning the behavioral model of low-level events.

2 Proof-of-Concept Implementation

Figure 1 shows detection of low-level events such as power and current surges, as detected by the power-management units (PMUs) that are positioned in key locations over the heterogeneous power grid. While such events may point to a low-level intrusion, the false-positives rate is high. In addition, a sophisticated attacker may coordinate intrusions at different points in the network in such a way that an attempt to filter-out the false-positives will also filter the actual intrusion.

In order to create a higher level of abstraction that represents unexpected changes in behavior of low-level events, a deep learning neural network is used in order to learn a model of low-level events, as illustrated in Figure 2 and Figure 3. Use of a neural network for this purpose is different from using one to detect anomalies as-is, which is an established research area as well [1, 2].

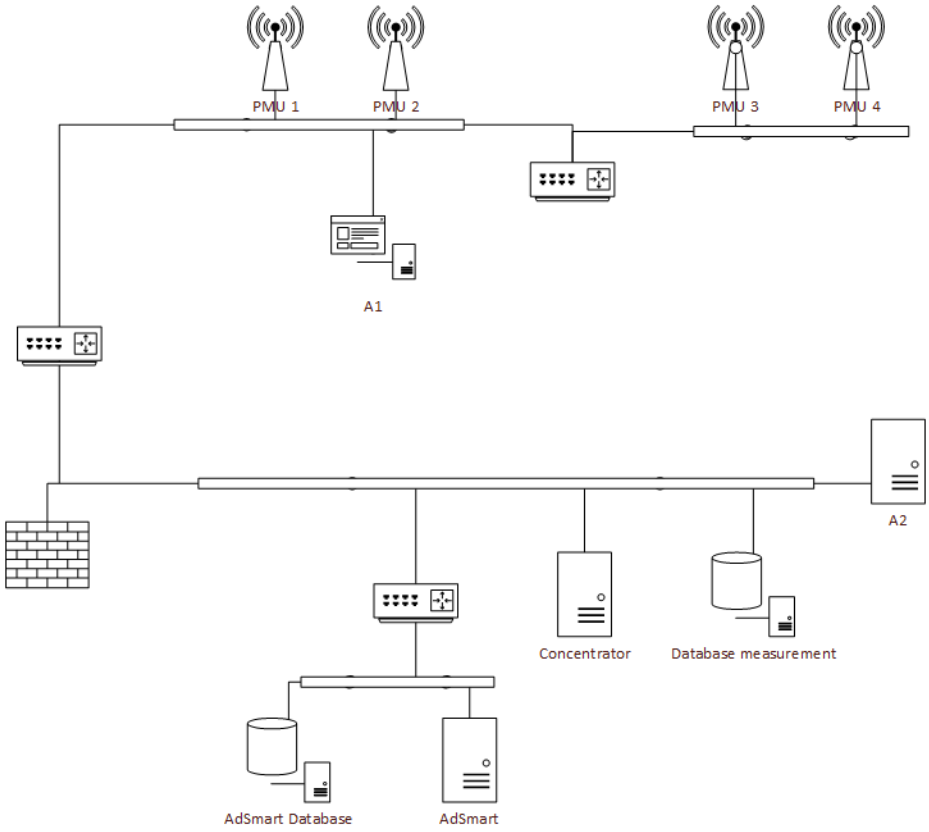


Fig. 1. Low-level events are detected from power management units (PMUs). AdSmart is the high-level events modeler described in this paper, whereas A1 and A2 are algorithms that detect low-level events.

The proof-of-concept is evaluated on a simulated power grid, as illustrated in Figure 4. Figure 5 shows a GUI that was implemented for testing the high-level events detection.

3 Conclusions

I have presented a POC for detecting high-level anomaly events in utility networks. The market for such a system could be power plants SCADA cybersecurity solutions, oil/gas pipelines control, and others. However, a test on real data is necessary. Moreover, deep learning has limited applicability for the approach, and advanced automatic learning methods like genetic programming are expected to yield more promising results.

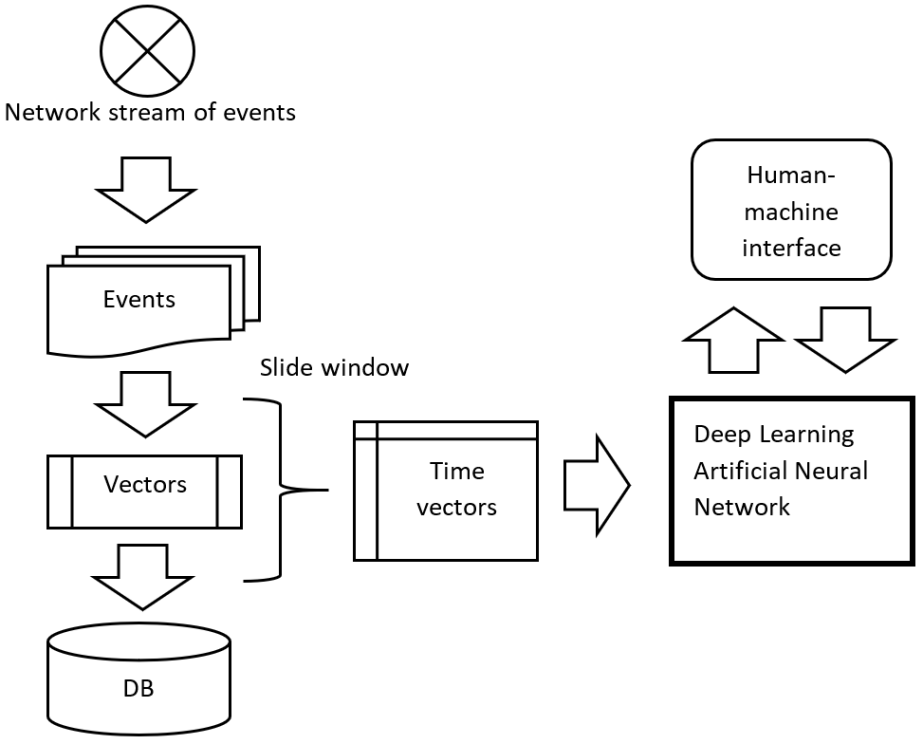


Fig. 2. Neural network learns a model of low-level events.

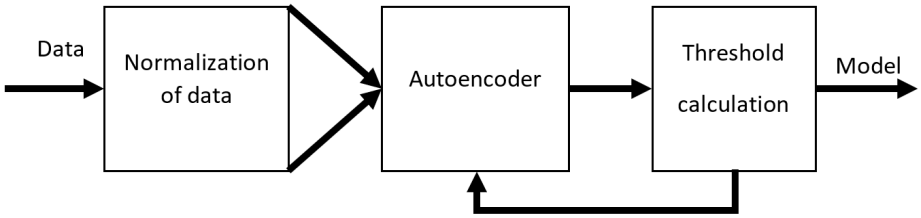


Fig. 3. Data flow for creating a model of low-level events.

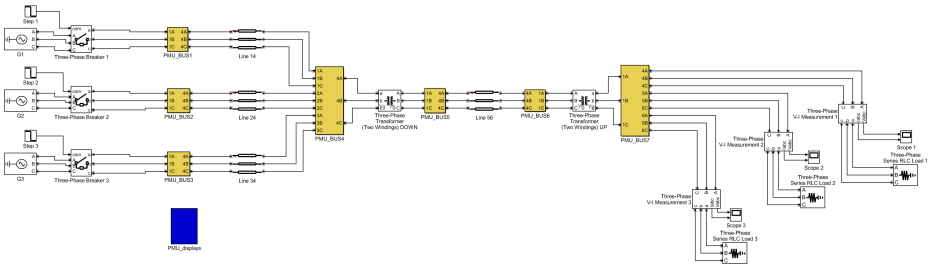


Fig. 4. A simulated power grid.

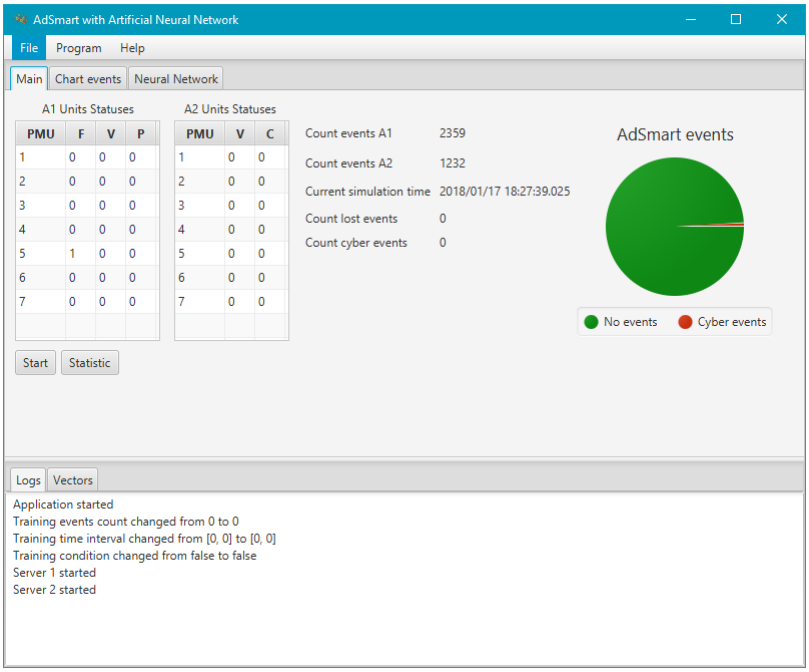


Fig. 5. A graphical user interface for the POC.

Acknowledgments

The project described herein was implemented by M.Sc. student Mr. Yvgeniy Dranko.

References

1. Kosek, A.M.: Contextual anomaly detection for cyber-physical security in smart grids based on an artificial neural network model. In: 2016 Joint Workshop on Cyber- Physical Security and Resilience in Smart Grids (CPSR-SG). vol. 00, pp. 1–6 (April 2016). https://doi.org/10.1109/CPSRSG.2016.7684103
2. Martinelli, M., Tronci, E., Dipoppa, G., Balducelli, C.: Electric power system anomaly detection using neural networks. In: Negoita, M.G., Howlett, R.J., Jain, L.C. (eds.) Knowledge-Based Intelligent Information and Engineering Systems. pp. 1242–1248. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
3. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Comput. Surv. **46**(4), 55:1–55:29 (Mar 2014). https://doi.org/10.1145/2542049