Poster: A Machine Learning Model Performance Improvement Approach to Detection of Obfuscated JavaScript-based Attacks

Samuel Ndichu
Graduate School of Engineering
Kobe University, Japan

Sangwook Kim
Graduate School of Engineering
Kobe University, Japan

Seiichi Ozawa
Center for Mathematical and Data Sciences
Kobe University, Japan

Abstract*—Obfuscation is rampant in both benign and malicious JavaScript (JS) codes. A JS code obfuscation generates a code that is obscure to the human eyes and undetectable to scanners, thereby hindering comprehension and analysis. This transformation significantly affects the performance of network and information security tools, such as Intrusion Detection System (IDS) and anti-virus software. Therefore, accurate detection of JS codes that masquerade as innocuous scripts is vital. The existing deobfuscation methods assume that a specific tool can recover an original JS code entirely. For a multi-layer JS code obfuscation, general tools realize a readable and formatted JS code, but some sections remain encoded. For the detection of such obfuscated codes, this study performs Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) by function redefinition using a JS code formatter, a Virtual Machine (VM), a JS code editor, and a python $int\_to\_str()$ function to facilitate feature learning by the FastText model, a machine learning model. The learned feature vectors are passed to SVM, a classifier model that judges the maliciousness of an obfuscated JS code. The proposed approach is envisioned to provide improved performance in obfuscated malicious JS codes detection. The detection performance improvement is evaluated using the Hynek Petrak's dataset for obfuscated malicious JS codes, the SRILAB, and the Majestic Million service top 10,000 websites dataset for obfuscated benign JS codes. We then compare the performance of the FastText model to Paragraph Vector models on the detection of DUD-preprocessed obfuscated malicious JS codes. Our experimental results show that the proposed DUD-preprocessing for obfuscated JS codes enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes compared to feature learning on regular obfuscated JS codes.

***Index terms*—** Deobfuscation, Unpacking, Decoding, Obfuscated JavaScript, Multi-layer JavaScript Obfuscation, JavaScript-based Attacks, FastText, Machine Learning

ACKNOWLEDGMENT

This research was achieved by the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Scientific Research (B) 16H02874 and the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

REFERENCES

[1] S. Kaplan, B. Livshits, B. Zorn, C. Siefert and C. Cursinger, *"NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"*, Microsoft Research Technical Report, MSR-TR-2011, 57, 2011, Pp.1-11.

[2] W. Xu, F. Zhang and S. Zhu, *The power of obfuscation techniques in malicious JavaScript code: A measurement study*, 7th International Conference on Malicious and Unwanted Software (MALWARE), IEEE, 2012, Pp.9-16.

[3] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik and E. Weippl, *Protecting Software through obfuscation: Can it keep pace with progress in code analysis*, ACM Computing Surveys (CSUR), 49, 1, 2016, Pp.1-40.

[4] S. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor and T. Parekhji, *A study and review on code obfuscation*, World conference on futuristic trends in research and innovation for social welfare (WCFTR'16), 2016, Pp.1-6.

[5] P. Skolka, C. Staicu and M. Pradel, *Anything to hide? Studying minified and obfuscated code in the web*, In the Proceedings of the WWW, San Francisco, CA, USA, ACM, 4, 2019, Pp.1-11.

[6] H. Petrak, *Javascript Malware Collection - A collection of almost 40.000 Javascript malware samples*, In: https://github.com/HynekPetrak/javascript-malwarecollection, Accessed on August 2019.

[7] V. Raychev, P. Bielik, M. Vechev and A. Krause, *Learning Programs from Noisy Data*, In Proceedings of the 43nd Annual ACM SIGPLAN-SIGACT Symposium on POPL, New York, NY, USA, ACM, 2016, Pp.761—774.

[8] The and Majestic and Million, *The million domains we find with the most referring subnets*, In: https://majestic.com/reports/majestic-million, Accessed on August 2019.

[9] P. Bojanowski, E. Grave, A. Joulin and T. Mikolov, *Enriching word vectors with subword information*, TACL, arXiv:1607.04606, 5, 2017, Pp.135-146.

[10] A. Joulin, E. Grave, P. Bojanowski and T. Mikolov, *Bag of Tricks for Efficient Text Classification*, In Proceedings of the 15th Conference of the EACL, Short Papers. Valencia, Spain, 2017, Pp.427-431.

[11] B. Wang, A. Wang, F. Chen, Y. Wang and C. C. J. Kuo, *Evaluating Word Embedding Models: Methods and Experimental Results*, APSIPA Transactions on Signal and Information Processing, arXiv:1901.09785, E19, 2019, Pp.1-13.

[12] T. Serafim and T. Kachalov, *JavaScript Obfuscator Tool*, A free and efficient obfuscator for JavaScript (including ES2017), In: https://obfuscator.io/, Accessed on August 2019.

[13] E. Lielmanis and L. Newman, *Online JavaScript Beautifier (v1.10.2), JavaScript and HTML, make JSON/JSONP readable*, In: https://beautifier.io/, Accessed on November 2019.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

A Machine Learning Model Performance Improvement Approach to Detection of Obfuscated JavaScript-based Attacks

NDSS SYMPOSIUM

KOBE UNIVERSITY

Samuel Ndichu, Sangwook Kim, Seiichi Ozawa
Kobe University

Abstract

- Obfuscation generates a JS code that is **obscure** to the human eyes and **undetectable** to scanners. JS code obfuscation aims to **hinder comprehension** and **analysis.** This transformation significantly affects the performance of network and information security tools, such as Intrusion Detection System (IDS) and anti-virus software. Therefore, accurate detection of JS codes that masquerade as innocuous scripts is vital.
- The existing deobfuscation methods for obfuscated malicious JS codes assume that a specific tool can recover an original JS code entirely. General tools realize a readable and formatted JS code, but **some sections remain encoded**. For detection of such obfuscated codes, this study performs Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) by function redefinition using a JS code formatter, a Virtual Machine (VM), a JS code editor, and a python *int_to_str()* function to facilitate feature learning by the FastText model. SVM, a classifier model, judges the maliciousness of an obfuscated JS code. The proposed approach is envisioned to provide improved performance in the detection of obfuscated malicious JS codes.

JS code obfuscation

- JS code obfuscation advantages:
  - Proprietary code protection.
  - Curbing reverse engineering.
  - Performance optimization.
  - Code compression.

The string "New User" in hello("New User") from the original JS code below is replaced to "var _0x74f5", a call function that retrieves its value at runtime, in the obfuscated JS code.

</> Javascript Obfuscator

```
function hello(name){
   console.log("Hello, " + name);
}
hello("New user");
```

Original JS code

```
var
_0x74f5=["\x48\x65\x6C\x6C\x6F\x2C\x20","\x6C\x6F\x67","\x4E\x65\x77\x20\x75\x73\x65\x72"];function hello(_0xe170x2){console[_0x74f5[1]](_0x74f5[0]+_0xe170x2)}hello(_0x74f5[2])
```

Obfuscated JS code

An example using hexadecimal to implement encoding.

JS code deobfuscation

- Obfuscated JS code analysis and formatting to make it readable again and uncover its true functionality.
- Tools to analyze obfuscated JS code: such as, Dirty Markup, Online JS code beautifier, Dan's Tools JS code formatter, and JSNice.

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('3 0(1){2.4("5, "+1)}0("7 6");',8,8,'hello|name|console|function|log|Hello|user|New'.split('|'),0,{}))
```

Original JS code

```
'use strict';
/**
 * @param {string} name
 * @return {undefined}
 */
function hello(name) {
  console.log("Hello, " + name);
}
hello("New user");
```

Deobfuscated JS code

The **eval()** function in the original JS code attempts to run the packed JS code.

Method

※ **Objective** – Performance improvement for detection of obfuscated malicious JS codes using FastText.

Steps to deobfuscate, unpack and decode an obfuscated JS code

Obfuscated JS code

1. Deobfuscate an obfuscated JS code – using a JS code beautifier, formatter: *These tools make JS code look pretty, readable, easier to edit and analyze.*

Beautified / Formatted JS code

2. Unpack a packed JS code – using a Virtual Machine (VM) and a JS code editor:
- Strip the script tags; *JS_code* = '''*eval(function(p,a,c,k,e,d)...obfuscated_JS_code...)*'''
- Replace the eval() function with console.log().
- Parse the packed JS code; *Unpacked_ JS_code = eval('unpack' + JS_code[JS_code.find('}(')+1:-1])*

Unpacked JS code

3. Decode an encoded JS code – using an Int_to_str() function in python:
- Implement Int_to_str() function in python.
- Parse to extract the function arguments – using a VM and a JS code editor.

Plain JS code

JS code deobfuscation, unpacking and decoding

Obfuscated JS Codes → JS code Formatter / Beautifier → Deobfuscated JS Codes → JS code Unpack / Decode → Plain JS Codes → JS code Parser / Syntactic Analyzer

Feature Learning and Classification

Train FastText model on deobfuscated, unpacked and decoded JS codes

Deobfuscated, Unpacked / Decoded AST-JS → Feature Learning Using FastText Model → SVM Classifier Training → Benign JS code / Malicious JS code

JS – JavaScript
ASTs – Abstract Syntax Trees
AST-JS – AST form of JS code

The FastText model:

- Character n-gram vectors represents each word x.
- Scoring function f takes into account a word internal structure.
- Character n-gram for *encode* with n=3: < en; enc; nco; cod; ode; de > and < encode >.

- For a dictionary of size G n-gram vectors, $G_x \subset \{1, \ldots, G\}$ gives the set of ngram vectors in x. The scoring function f is given by,

$$f(x,c) = \sum_{g \in G_x} \mathbf{Z}_g^{\top} \mathbf{X}_c$$

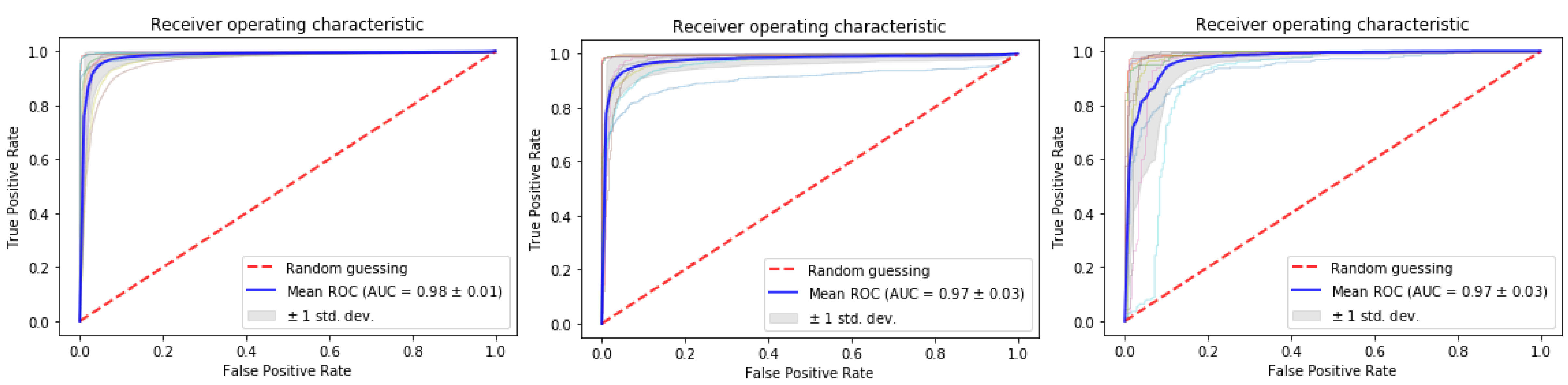
- Where $\mathbf{Z}_g$ is the vector representation for each n-gram g and $\mathbf{X}_c$ is the context.

model_JS:wv:most_similar("*encode*") – '*encodeurl*'; 0:91, '*encodeuri*'; 0:89, '*htmlencode*'; 0:89, '*enc_str*'; 0:87, '*decodeuri*'; 0:83.

Performance Evaluation

Model	Precision	Recall	F1-score
	Obfuscated JS code Dataset		
FastText	94.27%	95.11%	94.59%
PV-DBoW	94.12%	92.83%	93.31%
PV-DM	93.13%	93.51%	92.89%
	Deobfuscated JS code Dataset		
FastText	99.48%	99.31%	98.73%
PV-DBoW	98.39%	98.41%	98.01%
PV-DM	98.37%	98.02%	98.19%

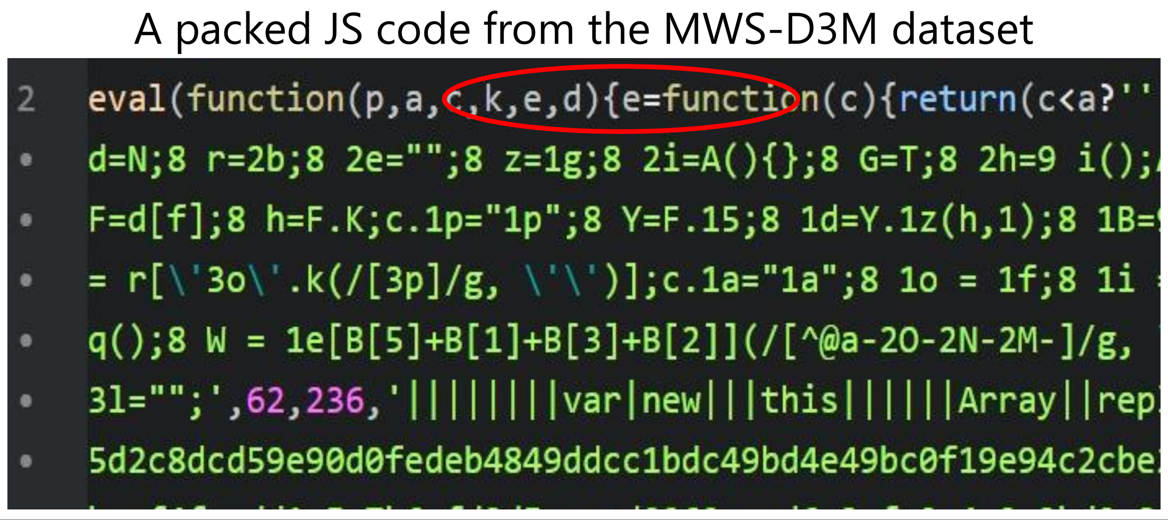
True positive and false positive rate



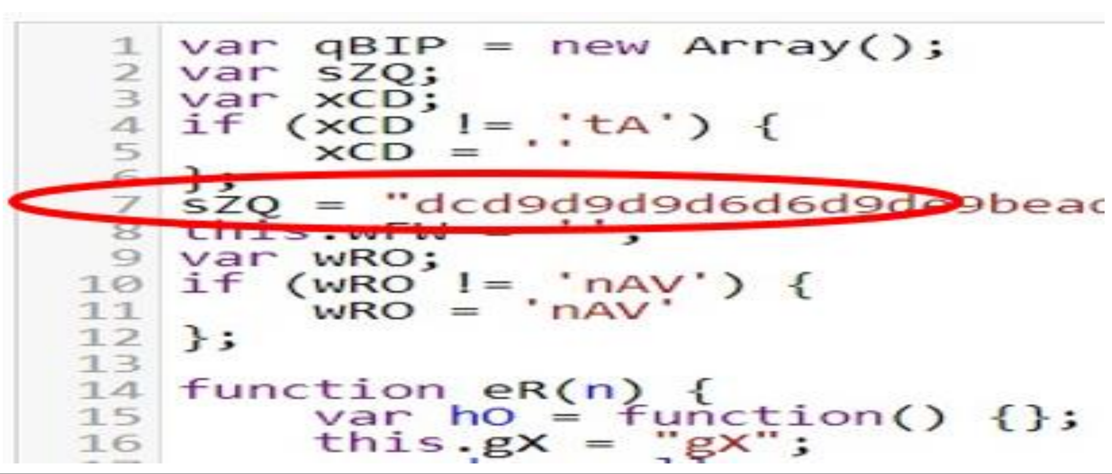
FastText PV-DBoW PV-DM

A hard-to-deobfuscate JS code example

A packed JS code from the MWS-D3M dataset



Using a JS code beautifier or formatter



- JS-based attacks frequently use obfuscation to:
  - Camouflage their malicious intentions.
  - Preserve the overall code behavior.
  - Evade detection.
- The FastText model learns better and reliable vector representations for DUD-preprocessed obfuscated malicious JS codes.