An Analysis of Obfuscated Malicious JavaScript

Nicholas Whitefield and Jorden Hansen
University of Alberta
Canada
{whitefie,jorden1}@ualberta.ca

Abstract

JavaScript's popularity has grown rapidly in recent years, showing up in all areas of web development [2]. With the rise of JavaScript, there is also an increase in malicious JavaScript code. We aim to identify some weaknesses of current malware detection tools, as well as identify how different methods of obfuscation perform on these tools. By conducting a survey on a sample of 1000 malicious JavaScript files, we observe how the obfuscation methods perform. We selected 6 different obfuscation methods, which we applied both individually as well as all possible pairings, totalling 21000 obfuscated JavaScript files. After removing tools we determined to be too ineffective at detecting malicious files, we were left with 18 tools which we used to analyse the obfuscated files.

Keywords JavaScript, Obfuscation, Malware, Program Analysis

1 Introduction

Malware is becoming more and more of a societal issue, with the rise of ransomware attacks and large scale data breaches, there is a greater need for stronger malware detection methods. One of the many challenges with malware analysis is that malware can be modified to be intentionally difficult to analyse due to obfuscation. Therefore, it is necessary for malware detection tools to be able to see through obfuscation attempts. In this paper we explore six different types of obfuscation methods and observe how they impact the detection rates among a variety of different analysis tools. In our research, we focus exclusively on malicious JavaScript files, we do this for two main reasons: the first being that it was the most abundant form of malware we found, and the second being that JavaScript is supported by a large number of tools and libraries, which allowed us to focus more on getting results and less on building the tools required to create our dataset.

1.1 Obfuscation

Obfuscation is the process of converting code into a more convoluted version of itself, without introducing any changes to the intended behaviour. From the perspective of an attacker, obfuscation is a useful technique as it weakens a victim's ability to understand and analyse the malicious program. With this in mind, techniques have been proposed that aim to detect malware by first detecting obfuscation and using that as a measure of malice [1]. However, we cannot simply consider any obfuscated program to be malicious, as there exists valid non-malicious use cases for obfuscation in both academia and industry [1]. A notable application for such obfuscation is to prevent thieves from reverse engineering proprietary algorithms, which can be very costly to research and develop and difficult to pursue legally when theft does occur [3]. Obfuscation applied with non-malicious intent is referred to as *benign obfuscation* [10].

2 Implementation

There are a number of tools that we used to create and gather our results. For the obfuscation of our dataset, we used an open source JavaScript obfuscator which can be downloaded from GitHub [5]. The tool provides a command line interface for applying various obfuscation methods to JavaScript files. The obfuscator supports 18 obfuscation methods that can be applied, however, some of these methods are more tailored towards benign obfuscation which we eliminated from consideration. Seeing as one of the areas we were interested in exploring was how these methods interact with one another, we decided to reduce the set of methods further to decrease the number of potential method combinations to a manageable size. In total, we selected six methods which we choose based on our rough predictions of their effectiveness. These six methods, along with all 15 possible pair combinations of these options were then applied to our dataset and analysed. These methods are highlighted below.

- *Control Flow Flattening*: this obfuscation method "flattens" the control flow of a program, making it harder to analyze the code. We used the default value of 0.75 for the obfuscation tool [5].
- *Dead Code Insertion*: this obfuscation method adds additional lines of code to the file in an attempt to make it harder to follow the actual code of the file [5]. We used the default value of 0.4 for the obfuscation tool. It is important to realize that dead code insertion does increase the size of the code. If it is important that the malware file stay small in size, this may not be the ideal obfuscation method.
- *Identifier names generation*: this method changes the identifier names in the code. It provides two options, hex and mangled. The hex option changes identifiers to be hex values, and mangled uses short values for names. [5]

Figure 1. Obfuscation with Transform Object Keys

```
// Input                  // Output
var object = {            var _0x5a21 = [
    foo: 'test1',             'foo',
    bar: {                    'test1',
        baz: 'test2'          'bar',
    }                         'baz',
};                            'test2'
                          ];
```

- *Self-Defending*: this option makes it so that the code is resilient to change [5]. Any attempt to rename or change the code structure will render the code unexecutable. This can make it much harder to analyze code since it cannot be reformatted in anyway.
- *String Array Encoding*: this obfuscation method encodes string literals in the code to be base64 or rc4 [5]. In our case we used base64. This decision was due to the fact that using rc4 would lead to a greater slow down in execution time over base64. We do not want to change runtime too much for the malware.
- *Transform object keys*: this method changes the way that JavaScript objects are stored [5]. This is illustrated in figure 1.

For the analysis of our dataset, we used VirusTotal's virus scan API which allowed us to upload files and receive a scan report which contained results from all the tools VirusTotal used in the scan [8]. In total, there are over 70 tools that VirusTotal can run on submitted files, however, each scan does not necessarily use the same set of tools. One of the main benefits VirusTotal provided was that it enabled us to automate the process of submitting files and retrieving results, allowing us to use a larger dataset of malware. By using VirusTotal, we were also able to abstract away from any particular tool which allowed us to gather data across a large set of tools.

In order to ensure that the obfuscation methods being applied were not simply destroying the original source files, we used a syntax-checking library which checked for any syntax errors in our obfuscated files [7]. The library itself has over 400 000 weekly downloads on npm and is still being maintained, so we assume its results are reliable. Throughout our obfuscation process, we did not observe a single occurrence of invalid-syntax, so we can also reasonably assume that the obfuscation tool does not output syntactically invalid JavaScript.

We sampled malicious JavaScript files from a GitHub repository [4]. The repository provides over 40 000 malware samples, but since each obfuscation method creates a new file that we must manage, we decided to only use the files provided in the 2015 folder, totaling to 1000 files. Our obfuscated files dataset along with the original samples are available in our project GitHub repository [9], as well as the raw results of the VirusTotal scans and our code for analysing these results.

The general workflow of our process is shown in figure 2. For each obfuscation technique, we would upload the batch of 1000 obfuscated JavaScript files to VirusTotal then run our analysis scripts on the returned scan results.

3 Evaluation

In this study, we had two research questions that we set out to answer.

3.1 RQ1: What is the effectiveness of each obfuscation technique?

As we briefly mentioned earlier, when we scanned the original (unobfuscated) files through VirusTotal we found that many of the tools performed inadequately. These tools either had extremely low detection rates or in some cases failed to detect even a single file. It seemed unlikely that a tool would perform better once we obfuscated the files, therefore we decided that these tools should be removed from our toolset as they would add very little value to our results. We filtered out any tools that had a detection rate less than 20%. Once we began analysing our obfuscated dataset, we noticed that the detection accuracy of many tools dropped significantly. We were more interested in observing the results of competent tools so we needed to build a *whitelist* of tools that we considered to be worthwhile. We ranked tools based on the number of obfuscation methods that the tool was able to detect, using a threshold of 20%. Meaning if the tool had a detection accuracy greater than 20% for a given method, we incremented that tools rank by one. From that point on the rest of our analysis used only results from the 18 highest ranked tools, we refer to this set as the *official-whitelist* of tools (figure 1).

After analyzing the results, we learned that control flow flattening is by far the most effective single obfuscation method, with an average tool detection rate of only 37.2%. The next best method was found to be string array encoding with a detection rate of 67.98%. The least effective method was dead code insertion at 81.87%. This still gives us a detection rate better than the original files without obfuscation which was 97.13%.

When we pair two obfuscation methods, the best two methods use both control flow flattening and string array encoding. This is not surprising as these were the top two methods individually. One interesting observation is that when control flow flattening is paired with dead code insertion, it performs better than control flow flattening paired with transform object keys, which has detection rates at 35.31% and 36.37% respectively. Our worst two method obfuscation was when we combined dead code insertion with

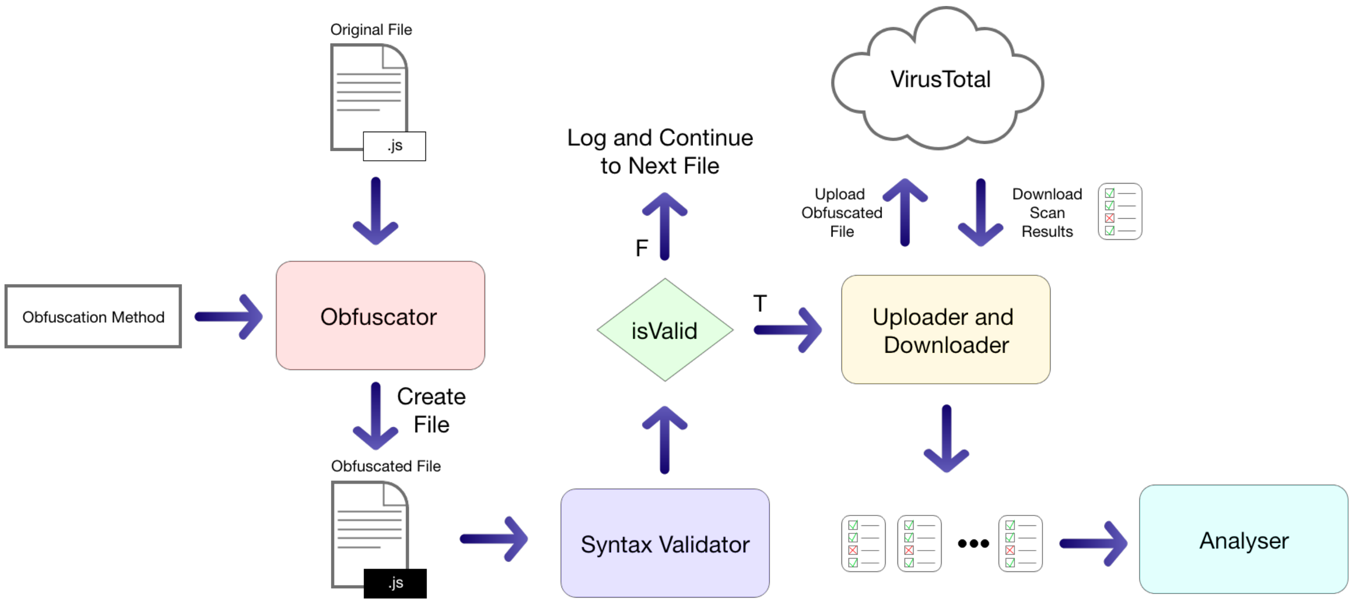


Figure 2. Our Obuscation Pipeline

Table 1. Obfuscation Detection Accuracy for Different Tool Filters (In Percent)

filter	orig	CFF	DCI	ING	SD	SAE	TOK
nofilter	62.26	12.16	26.78	27.01	22.91	23.60	28.91
originalFiles-whitelist	87.51	17.25	38.00	38.35	32.52	31.03	41.68
official-whitelist	97.13	37.20	81.87	79.79	71.49	67.98	81.50

transformation of object keys, which performed slightly better than each does on their own, at 81.06%.

3.2 RQ2: How well do the tools perform individually and how do they compare?

Our best performing tool on single method obfuscation was Kaspersky which had an average detection rate of 96.26%. The next best tool was ZoneAlarm which had a detection rate of 96.25%, putting it very close to Kaspersky. The worst tool we had was McAfee-GW-Edition with an average detection rate of 41.38%. The majority of the tools we analysed performed the worst when control flow flattening was used to obfuscate. The mean detection rate across tools is 73.76%

What is interesting though is that our three best tools (Kaspersky, ZoneAlarm, and NANO-Antivirus) all perform the worst when identifier name generator is used as an obfuscation method. They still perform better than other tools, but this was the only method which had a consistent decrease in overall detection rate (A decrease of 6%). A few tools, Qihoo-360, Ikarus, Microsoft, and McAfee-GW-Edition all perform the worst when string array encoding is used. Symantec performs very well as a tool, placing 4th, even though it cannot detect any malware that is obfuscated with self-defending. This result is possibly evidence of Symantec trying to alter the code in some way, triggering the self-defending, which then makes the tool fail to detect any malware. If we ignore the self-defending cases, we see that Symantec has a detection rate very comparable to NANO. Overall, we can see that some tools perform worse on certain obfuscation techniques, while others have little to no effect from that method.

Moving onto cases where two obfuscation methods are used, We see that ZoneAlarm and Kaspersky actually have the same average detection rate, at 95.04%. The worst tool is again McAfee-GW-Edition, at a mean detection rate of 18.63%. The average across all tools is 58.37%, this is dragged down mostly by control flow flattening with any other method, where the majority of the tools perform rather poorly. A few interesting results showed up here though. Ikarus has a detection rate of 25.00% when string array encoding is used with self-defending, which is exceptionally high when compared to other obfuscation methods that also used string array encoding (all others falling below 5%). Even just string array encoding alone has a lower detection rate than when paired with self-defending. TrendMicro performs exceptionally bad when self-defending and identifier name generator are used together, scoring 0.30%, its next lowest detection rate is 18.88% on control flow flattening with string array encoding. There must be something interesting going on with this interaction because individually it scores 51.8% on self-defending and 76.3% on identifier name generator.

McAfee-GW-Edition scores a 0% on code obfuscated with control flow flattening and string array encoding, this is only interesting because the only other case of a detection rate of 0% is with Symantec and self-defending code, which it likely cannot handle. While McAfee is able to detect both methods individually, McAfee is unable to detect any malicious javaScript obfuscated with this pairing.

Overall, We can see that our top ranked tools are rather resistant to most obfuscation methods, never dropping below 90% for Kaspersky and ZoneAlarm on any obfuscation method and only having a noticeable dip when identifier name generator is used, regardless of what other obfuscation method is used.

4 Limitations

The results of our study have limitations due to a number of assumptions that we made throughout the study:

- **Is the Malware Malware?**: The malicious JavaScript samples that we used are supposedly malicious but without manual inspection of each file it would be very difficult to verify this, however, we are confident that at least 98% of the files are malicious based on the scan results from VirusTotal.
- **Code Destruction**: We provide a safety mechanism to detect if the obfuscator ever 'destroys' the input files by using a syntax checking library, but this does not account for potential program-behaviour-modification by the obfuscator.
- **Randomness**: Since the obfuscator tool used some amount of randomness when performing obfuscation, the small differences in some of our results could simply be due to this randomness. This could be solved by seeding the obfuscation tool, which we only realized after collecting all our results.
- **Verifiability**: While most of our process is automated and can be replicated, we cannot guarantee identical results due to our inherent dependency on VirusTotal. The tools that VirusTotal uses to generate its scans are likely updated often, also VirusTotal itself likely includes and removes tools from its toolset. While we do not think this would drastically alter our results, it could lead to some deviation in accuracy for anyone replicating our study.

5 Related Work

5.1 Similar Research in JavaScript Obfuscation

Wei Xu, Fangfang Zhang, and Sencun Zhu have also done similar research in this area, however they do not use the same tools and obfuscation methods that we used [10]. In their paper, they use 4 obfuscation methods: randomization, encoding, data obfuscation, and logic structure obfuscation. They introduce randomization by applying white space, comments and renaming variables and functions [10]. This is similar to the identifier name generator that we use. Their data obfuscation is string computations, keyword substitution and number computations [10]; we do not have a comparable obfuscation technique to this one. The last obfuscation technique they test is encoding, which is similar to our string array encoding, however they only use hexadecimal, ASCII, or unicode, whereas the method we used is base64 encoded. They mention logic structure obfuscation, but provide no results for it [10].

5.2 Detection using Machine Learning:

Research has also been done in looking at the application of machine learning to detect obfuscated Malicious JavaScript [6]. In this paper, the authors propose a classification based method for detecting the presence of obfuscation itself, which they consider to be a strong indicator for malware. A model using the classification based approach is advantageous because it can still detect malicious JavaScript even if it has never seen that particular instance before [6]. They find that classification is a viable method for building accurate detection tools but only if the feature vectors are carefully constructed. There are also drawbacks with classification, most notably the classifier can be susceptible to false positives [6]. Packing is the most likely obfuscation technique to create false positives, since packing is often used in a benign way to compress data sent from servers to clients [6]. Another weakness of classifiers is their use of feature vectors; if any attacker learns this feature vector they can modify their code to less exemplify those features, in which case the model would need to be retrained or new features would need to be identified [6].

6 Future Work

There are a few areas of future work we have considered for this project:

- For starters, we could apply additional obfuscation methods and see how they interact. Also, we only looked into single and paired methods, but it would be interesting to see results from applying 3 or more methods. There could also be further work in analysing dead code insertion and control flow flattening with varying threshold values, and measure the effectiveness of such thresholds.
- We could also look into incorporating other languages into our analysis.
- Many parts of our project are automated but some areas rely on user interaction, ideally we could increase the level of automation to streamline analysis.
- Another area of interest would be looking more closely at the files themselves to see if certain ones are more susceptible to detection than others or if any avoid

detection altogether. We are also interested in looking more into the tools themselves to possibly better explain some of the outliers we observerd.

7 Conclusion

Through this study, we found that the tools we analysed generally fell into one of three categories. Category one is the tools that perform well across all obfuscation results, this is Kaspersky, ZoneAlarm, and NANO. Category two is tools that perform well across all obfuscation results except control flow flattening, this is the biggest group. Category 3 is the tools that have the most trouble with string array encoding.

We also found the best overall obfuscation method to be control flow flattening and the worst to be dead code insertion, followed closely by transformation of object keys. The data collected points out some of the weaknesses in detection software, showing where some need improvement, but also highlighting what each tool handles well. However, this also shows that if a malicious party knows what type of detection software they are up against, they can meaningfully pick an obfuscation method targeted towards that tool.

References

[1] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, and Youki Kadobayashi. 2012. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, Fukuoka, Japan, 344–351. https://doi.org/10.1109/WAINA.2012.140

[2] David Cassel. 2016. JavaScript Popularity Surpasses Java, PHP in the Stack Overflow Developer Survey. Retrieved December 6, 2019 from https://thenewstack.io/javascript-popularity-surpasses-java-php-stack-overflow-developer-survey/

[3] Christian Collberg, Clark Thomborson, and Douglas Low. 1996. *A Taxonomy of Obfuscating Transformations*. Technical Report. Department of Computer Sciences, The University of Auckland. https://www.researchgate.net/publication/37987523_A_Taxonomy_of_Obfuscating_Transformations

[4] HynekPetrak. 2016. javascript-malware-collection. Retrieved November, 2019 from https://github.com/HynekPetrak/javascript-malware-collection

[5] javascript-obfuscator 2019. Retrieved November, 2019 from https://github.com/javascript-obfuscator/javascript-obfuscator

[6] Peter Likarish, Eunjin (EJ) Jung, and Insoon Jo. 2009. Obfuscated Malicious Javascript Detection using Classification Techniques. In *2009 4th Int. Conf. on Malicious and Unwanted Software (MALWARE)*. IEEE, Montreal, QC, Canada, 47–54. https://doi.org/10.1109/MALWARE.2009.5403020

[7] syntax-checker 2019. Retrieved November, 2019 from https://www.npmjs.com/package/syntax-error

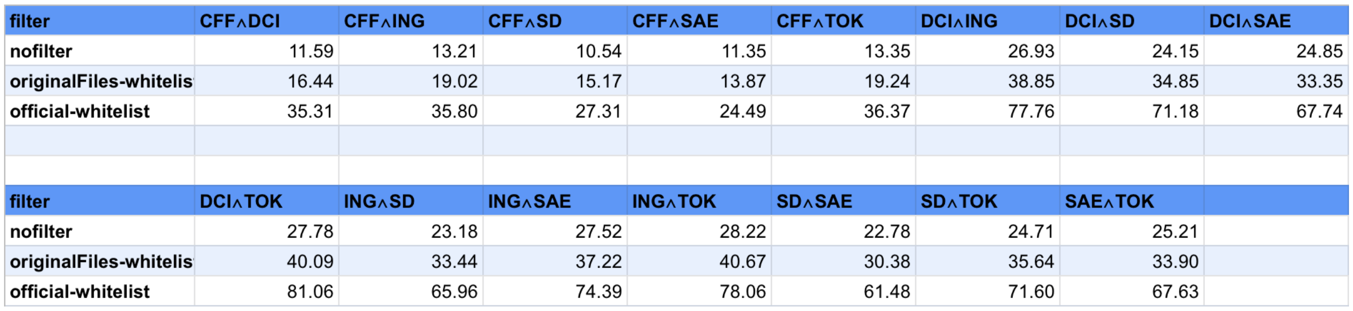
[8] VirusTotal 2019. Retrieved December 1, 2019 from https://developers.virustotal.com/reference

[9] Nicholas Whitefield and Jorden Hansen. 2019. Analysis Repository. https://github.com/npwhite/cmput497-project

[10] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th Int. Conf. on Malicious and Unwanted Software*. IEEE, Fajardo, PR, USA, 9–16. https://doi.org/10.1109/MALWARE.2012.6461002

A Tables

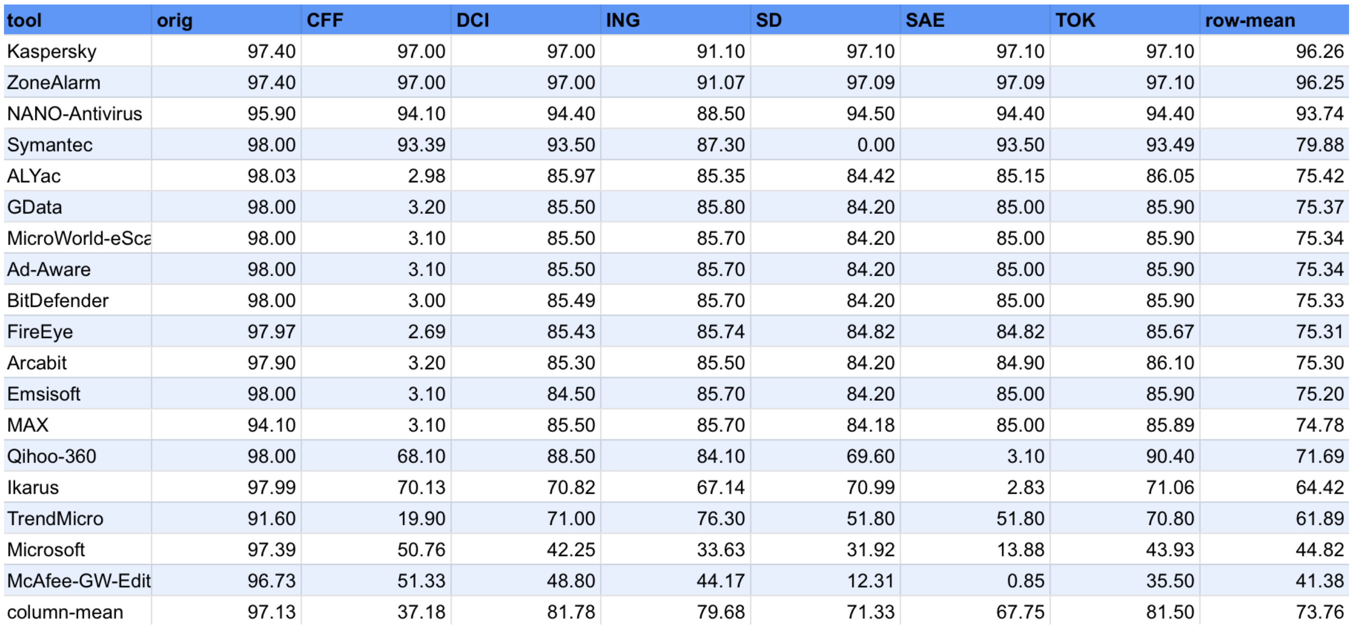
This appendix contains the tables that were too large to fit into the paper.



filter	CFF∧DCI	CFF∧ING	CFF∧SD	CFF∧SAE	CFF∧TOK	DCI∧ING	DCI∧SD	DCI∧SAE
nofilter	11.59	13.21	10.54	11.35	13.35	26.93	24.15	24.85
originalFiles-whitelis	16.44	19.02	15.17	13.87	19.24	38.85	34.85	33.35
official-whitelist	35.31	35.80	27.31	24.49	36.37	77.76	71.18	67.74

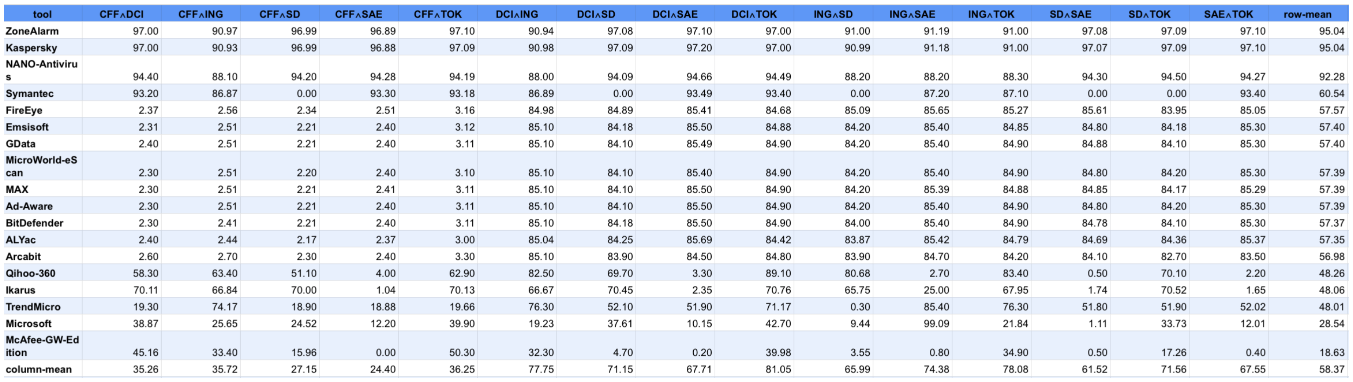
filter	DCI∧TOK	ING∧SD	ING∧SAE	ING∧TOK	SD∧SAE	SD∧TOK	SAE∧TOK	
nofilter	27.78	23.18	27.52	28.22	22.78	24.71	25.21	
originalFiles-whitelis	40.09	33.44	37.22	40.67	30.38	35.64	33.90	
official-whitelist	81.06	65.96	74.39	78.06	61.48	71.60	67.63	

Figure 3. Comparing Paired Obfuscation Methods Across Different Toolsets



tool	orig	CFF	DCI	ING	SD	SAE	TOK	row-mean
Kaspersky	97.40	97.00	97.00	91.10	97.10	97.10	97.10	96.26
ZoneAlarm	97.40	97.00	97.00	91.07	97.09	97.09	97.10	96.25
NANO-Antivirus	95.90	94.10	94.40	88.50	94.50	94.40	94.40	93.74
Symantec	98.00	93.39	93.50	87.30	0.00	93.50	93.49	79.88
ALYac	98.03	2.98	85.97	85.35	84.42	85.15	86.05	75.42
GData	98.00	3.20	85.50	85.80	84.20	85.00	85.90	75.37
MicroWorld-eSca	98.00	3.10	85.50	85.70	84.20	85.00	85.90	75.34
Ad-Aware	98.00	3.10	85.50	85.70	84.20	85.00	85.90	75.34
BitDefender	98.00	3.00	85.49	85.70	84.20	85.00	85.90	75.33
FireEye	97.97	2.69	85.43	85.74	84.82	84.82	85.67	75.31
Arcabit	97.90	3.20	85.30	85.50	84.20	84.90	86.10	75.30
Emsisoft	98.00	3.10	84.50	85.70	84.20	85.00	85.90	75.20
MAX	94.10	3.10	85.50	85.70	84.18	85.00	85.89	74.78
Qihoo-360	98.00	68.10	88.50	84.10	69.60	3.10	90.40	71.69
Ikarus	97.99	70.13	70.82	67.14	70.99	2.83	71.06	64.42
TrendMicro	91.60	19.90	71.00	76.30	51.80	51.80	70.80	61.89
Microsoft	97.39	50.76	42.25	33.63	31.92	13.88	43.93	44.82
McAfee-GW-Edit	96.73	51.33	48.80	44.17	12.31	0.85	35.50	41.38
column-mean	97.13	37.18	81.78	79.68	71.33	67.75	81.50	73.76

Figure 4. Comparing individual Tools Across Single Obfuscation Methods



tool	CFF∧DCI	CFF∧ING	CFF∧SD	CFF∧SAE	CFF∧TOK	DCI∧ING	DCI∧SD	DCI∧SAE	DCI∧TOK	ING∧SD	ING∧SAE	ING∧TOK	SD∧SAE	SD∧TOK	SAE∧TOK	row-mean
ZoneAlarm	97.00	90.97	96.99	96.89	97.10	90.94	97.08	97.10	97.00	91.00	91.19	91.00	97.08	97.09	97.10	95.04
Kaspersky	97.00	90.93	96.99	96.88	97.09	90.98	97.09	97.20	97.00	90.99	91.18	91.00	97.07	97.09	97.10	95.04
NANO-Antivirus	94.40	88.10	94.20	94.28	94.19	88.00	94.09	94.66	94.49	88.20	88.20	88.30	94.30	94.50	94.27	92.28
Symantec	93.20	86.87	0.00	93.30	93.18	86.89	0.00	93.49	93.40	0.00	87.20	87.10	0.00	0.00	93.40	60.54
FireEye	2.37	2.56	2.34	2.51	3.16	84.98	84.89	85.41	84.68	85.09	85.65	85.27	85.61	83.95	85.05	57.57
Emsisoft	2.31	2.51	2.21	2.40	3.12	85.10	84.18	85.50	84.88	84.20	85.40	84.85	84.80	84.18	85.30	57.40
GData	2.40	2.51	2.21	2.40	3.11	85.10	84.10	85.49	84.90	84.20	85.40	84.90	84.88	84.10	85.30	57.40
MicroWorld-eScan	2.30	2.51	2.20	2.40	3.10	85.10	84.10	85.40	84.90	84.20	85.40	84.90	84.80	84.20	85.30	57.39
MAX	2.30	2.51	2.21	2.41	3.11	85.10	84.10	85.50	84.90	84.20	85.39	84.88	84.85	84.17	85.29	57.39
Ad-Aware	2.30	2.51	2.21	2.40	3.11	85.10	84.10	85.50	84.90	84.20	85.40	84.90	84.80	84.20	85.30	57.39
BitDefender	2.30	2.41	2.21	2.40	3.11	85.10	84.18	85.50	84.90	84.00	85.40	84.90	84.78	84.10	85.30	57.37
ALYac	2.40	2.44	2.17	2.37	3.00	85.04	84.25	85.69	84.42	83.87	85.42	84.79	84.69	84.36	85.37	57.35
Arcabit	2.60	2.70	2.30	2.40	3.30	85.10	83.90	84.50	84.80	83.90	84.70	84.20	84.10	82.70	83.50	56.98
Qihoo-360	58.30	63.40	51.10	4.00	62.90	82.50	69.70	3.30	89.10	80.68	2.70	83.40	0.50	70.10	2.20	48.26
Ikarus	70.11	66.84	70.00	1.04	70.13	66.67	70.45	2.35	70.76	65.75	25.00	67.95	1.74	70.52	1.65	48.06
TrendMicro	19.30	74.17	18.90	18.88	19.66	76.30	52.10	51.90	71.17	0.30	85.40	76.30	51.80	51.90	52.02	48.01
Microsoft	38.87	25.65	24.52	12.20	39.90	19.23	37.61	10.15	42.70	9.44	99.09	21.84	1.11	33.73	12.01	28.54
McAfee-GW-Edition	45.16	33.40	15.96	0.00	50.30	32.30	4.70	0.20	39.98	3.55	0.80	34.90	0.50	17.26	0.40	18.63
column-mean	35.26	35.72	27.15	24.40	36.25	77.75	71.15	67.71	81.05	65.99	74.38	78.08	61.52	71.56	67.55	58.37

Figure 5. Comparing individual Tools Across Paired Obfuscation Methods