Semantically Non-preserving Transformations for Antivirus Evaluation

Erkan Ersan[1], Lior Malka[2], and Bruce M. Kapron[1(✉)]

[1] Department of Computer Science, University of Victoria, Victoria, Canada
erkanersan@gmail.com, bmkapron@uvic.ca
[2] Faculty of Graduate Studies, University of Victoria, Victoria, Canada
lior34@gmail.com

Abstract. We relax the notion of malware obfuscation to include *semantically non-preserving transformations*. Unlike traditional obfuscation techniques, these transformation may not preserve original code behaviour. Using web-based malware we focus on transformations which modify abstract syntax trees. While such transformations yield syntactically valid programs, they may yield dysfunctional samples, so that it is not clear that this is a practical approach to producing detection-evading malware. However, by implementing an automated system that efficiently filters dysfunctional samples on a virtual cloud architecture, we show that such transformations are in fact practical. Using two simple transformations, we evaluated four antivirus products and were able to create many samples that evade detection, demonstrating that semantic-preserving obfuscation is not the only effective way to mutate malware.

1 Introduction

Recent data breaches at Target, Home Depot, JPMorgan Chase, Apple iCloud, and Sony (to name a few) highlight the constant pressure that cyber-attackers put on users and corporations alike [8]. Many attacks use malware, that is software with some form of malicious functionality [4], to steal financial information, intellectual property, and private data such as usernames and passwords. An *antivirus* is a tool for malware detection. Since many organizations rely on antiviruses for protection, it is important to evaluate antivirus effectiveness.

Obfuscation is a well known technique used by malware authors to create new malware *mutations* that evade detection. Obfuscation modifies code, while retaining its behaviour [6]. For example, in the context of HTML and Javascript, *renaming* might transform the code `payload=1; print(payload)` into `x=1; print(x)`, while *partitioning* might transform the code `str =‘‘abc’’` into `str =‘‘a’’+‘‘bc’’`. See [15] for more examples.

Christodorescu and Jha [5] proposed a methodology to evaluate antivirus products against obfuscated versions of known malware. They applied this idea

Research supported by Intel as part of the Collaborative Project “Automated Antivirus Evaluation via Malware Mutations”.

F. Cuppens et al. (Eds.): FPS 2016, LNCS 10128, pp. 273–281, 2017.
DOI: 10.1007/978-3-319-51966-1_18

to Visual Basic malware in Microsoft Office [5]. The same methodology was used in the context of Java malware in Android applications [13,16] and Javascript malware in HTML files [15]. Industry test labs follow the same approach [3].

Our research was motivated by considering the extension of [5] to the setting of browser-delivered malware ("drive-by downloads",) using HTML- and Javascript-based malware for Internet Explorer produced by Metasploit [9]. Manual experimentation indicated that malware obfuscated using traditional techniques (along the lines of e.g. [15]) were detected, but techniques which would not obviously produce semantically equivalent code (e.g. altering HTML elements, permuting lines of JavaScript) produced malware which still delivered its payload, but was no longer detectable. This experiment, and the analysis in Sect. 5, indicates that anti-malware tools should not be designed under the assumption that all mutations must result from obfuscation. For example, [10] proposed detection of malware mutations using a method that assumes mutant malware preserves semantics, an assumption challenged by our results.

Semantically non-preserving transformations have several disadvantages compared to obfuscation. With obfuscation, millions of mutations can be generated efficiently, they are all guaranteed to work, and if one class of transformations (e.g. variable renaming) bypasses detection, then most likely other mutations in this class also bypass detection. We show that, despite their disadvantages, semantically non-preserving transformations can be efficient and practical. Our main contribution is a cloud based system that automatically and efficiently generates malware mutations. The system is generic. It will work with any antivirus, any malware, and any transformation, including obfuscation. It even supports composition of transformations. Our system also scales linearly; doubling the size of the cloud reduces the computation time by half. We evaluated four antiviruses using two simple transformations, and yet were able to create many mutations that evade detection. Our system is different from script-based approaches used in prior work due to the specific challenges of our transformations. These challenges, and our solutions are discussed in detail in Sects. 3 and 4.

Our works focuses on obfuscation of what we might call *transporter code*, that is, the HTML/Javascript code which triggers an exploit allowing the delivery of a payload written in x86 code. While obfuscation of the payload is also a well-known technique for evading detection, our abstract-syntax based approach is not directly applicable to such obfuscations.

Related work. Evaluation of antivirus effectiveness via malware mutations have been considered in [5,11,13,14,16]. A formal framework for this method has been given as well (e.g., [7]). The reason for this evaluation method is that hackers evade detection by tweaking their malware. Unlike [5,13,16], which use obfuscation, we use transformations that do not preserve the semantics of the malware. We show that our transformations yield functional and undetectable variants. Also, unlike [5,13,16], where each component is automated, but not the system as a whole, our software is fully automated and non supervised. A recent industry report that evaluated eleven antiviruses against HTML malware [3] showed that

all antiviruses detected all HTML obfuscation. Our results show that evaluations which consider only transformations that preserve semantics are incomplete. In [12] a semantics-based approach to malware detection is proposed, including a definition of *non-conservative obfuscation* which is a generalization of our notion of semantically non-preserving transformation. We have not investigated the significance of our results in this broader framework.

2 Semantically Non-preserving Transformations

In formulating a notion of transformations which preserve semantics, we are faced with several choices. In the most general setting, we would need to formally model all the effects of executing a piece of code. This could involve modifications to state and side-effects involving not only memory, but also files, communication channels, etc. We will take a more practical perspective, tailored specifically to the setting of malware deployment and detection, and depends only on the ability of a piece of code to deliver a specified *payload*. See [12] for a more general approach to semantics-based notions of obfuscation.

We consider transformations $T : Code \times Aux_T \to Code$. In particular we have $T(m, r) = m'$ where m is the code to be modified and r is some auxiliary information and m' is the mutated code. The auxiliary information r depends on the transformation T. For example, in the case of variable renaming, Aux_T will consist a collection of variables and all strings to which the variables in question may be mapped. We then say that T is *semantically preserving* with respect to malware m if $T(m, r) = m'$ executes the same payload as m, for any $r \in Aux_T$. For example, obfuscation is always semantically preserving, whereas, in our above mentioned experiment, the transformation T that replaces `table` with various element names is semantically non-preserving, because there are r such that $T(m, r) = m'$ is not malware. On the other hand, if $T(m, r) = m'$ is malicious, it is not necessary that m' exploits a different vulnerability compared to m, or that m' executes other things beyond the payload of m. It only means that m' is allowed to be computationally different from m, as long as it executes the payload, and that T is allowed to output dysfunctional samples (that is, samples that do not execute the payload).

Because we admit transformations that output dysfunctional samples, we must address the issue of how generated samples may be used to test AV effectiveness. We describe our approach in Sect. 4 below, but in short we filter out dysfunctional samples by executing each sample and detecting whether it is able to deliver a payload. In practise this *post hoc* approach will produce samples that are not obtained using traditional obfuscation techniques.

3 The Generator

In this section we describe the software components that take an HTML file, possibly containing Javascript, and generate variants of this file. The variants

may or may not be malware. We collectively refer to these components as the *Generator*.

As discussed previously, we focus on a simple class of transformations, namely those which apply simple modifications to the abstract syntax tree (AST) of an HTML sample document. We implemented two transformations, *permute* and *subset*, both operating only on nodes that have children. its statements as children. By *permute* we mean reordering of the children, and by *subset* we mean removing some of the children. These transformations usually do not preserve the semantics of the original HTML file.

While our goal was to apply these transformations in as general and automatic a way as possible, it is clear that blindly applying these transformations to the entire AST results in a combinatorial explosion making the approach infeasible. In particular, for an AST with n nodes, subset will generate $O(2^n)$ mutations, while permute will generate $O(n!)$. We address this issue by specializing transformations to a distinguished subset of nodes, a technique we refer to as *per-node* transformation. Currently, we have not addressed the question of general strategies for assigning transformations to nodes. For our experiments we are doing this in an *ad hoc* fashion.

4 The Infrastructure

In this section we describe the software components that take original HTML files, transform them into new HTML variants, test whether the variants are *functional* (that is, they execute the payload), and if so, whether they are detectable as malware by various antivirus products. We collectively refer to these components as the *Infrastructure*.

This Infrastructure is realized via a *producer-consumer* model wherein a *producer* inserts jobs into a queue, blocking if the queue is full, and a *consumer* removes jobs from the queue and executes them, blocking if the queue is empty. Multiple producers and consumers can run concurrently, using database tables for queues and transactions for synchronization. While the use of virtualization in testing malware detection is not new, our automated concurrent infrastructure is unique. In practice, this means that we were able to perform fully automated tests involving several antivirus products and millions of malware variants.

In more detail, generator threads produce HTML variants for the functionality workers, who test whether HTML variants execute a payload. If a variant is functional, then the functional worker produces a job for the antivirus workers, who then test whether the variant is detectable or not.

In order to maintain automation, we need a simple test to determine functionality. We chose the creation by the malware of a text file on the desktop. A possible objection to this approach is that in a real-world setting such an action would be benign in terms of impact on the target system, and that more malicious effects could be detected by other components of a detection or intrusion protection system. With respect to the first objection, we note that the degree of control required to allow the malware to perform the file creation effect would

also allow it to perform more obviously malicious actions. This brings us to the second objection. Here we note that it is essential for all components of such a system to provide the highest degree of security possible. There is no guarantee that if any one component fails some other component will be able to compensate. Indeed, malware designers are able to combine evasion techniques to take advantage of weaknesses in any one detection component.

When a browser loads an HTML file, it stores it as a file in a temporary directory, and this triggers the detection mechanism of some antiviruses. However, due to the asynchronous nature of operating systems, quite often the antivirus detects the file as malware only after the browser has processed the HTML file and the malware has successfully executed. In such cases we consider the malware as being undetectable.

5 Experiments

We selected four popular antivirus products from the websites of the following companies: AVG, Kaspersky, McAfee, and Symantec. We have chosen not to disclose which product is susceptible to each method, and thus have randomly named them AV1, AV2, AV3, AV4. In all cases we downloaded the consumer grade version, and evaluated it automatically using our infrastructure.

We reviewed more than thirty Internet Explorer malware samples from Metasploit [9]. Unfortunately, not all were suitable for testing due to stability or compatibility reasons. Hence, only seven were chosen. We refer to these files, labelled $S1$ through $S7$, as the *originals*. All originals have a benign payload that creates a dummy text file on the desktop. Samples $S6$ and $S7$ did not parse with the parser of [1,2], while sample $S5$ had a payload that could not be configured. Thus, we could only experiment with the first four samples. Sample $S1$ was pure HTML, and the rest contained Javascript code. All samples were evaluated on Windows7 64-bit SP 1 and Internet Explorer 8.

Test Definitions. We define an HTML file to be *functional* if and only if the dummy is created when the HTML file is loaded by the browser. We note that, since malware can destabilize the operating system, we have to timeout our tests, which may incorrectly label a functional variant as non functional. However, this is not a concern. The important thing is that the opposite cannot happen.

Although we use the notion of *detection*, we follow industry practise [3] and measure *prevention*. i.e., whether the payload is prevented from executing. Prevention is stronger than detection because, once malware gets control, it can do anything, including disabling the antivirus, and detection in particular.

Our antivirus evaluation consisted of two tests. In the *static* test (denoted S) we invoked the antivirus from the command line, with the HTML file as input. In the *dynamic* test (denoted D) the browser loaded the HTML file from an external HTTP server. In the static test, some antiviruses remove the file before we even have a chance to invoke the antivirus from the command line. In such a case we treat the HTML file as being detected. Conversely, in the dynamic test,

if the dummy text file is created, then the payload has executed. Thus, even if the antivirus produces an alert, we consider the file as being undetected.

Analysis. The evaluation of the originals is given in Table 1. It shows four antiviruses evaluated against seven malware files, in both static and dynamic tests. We use 1 for *detected* and 0 for *undetected*. If the payload finished execution, but with high probability was later detected, then we denote it by 0*a*. Interestingly, AV3 can statically detect originals S3 through S6, and exactly those are detected in the dynamic test, but only after the payload executes. This provides a strong evidence that AV3 is signature based only, not using any runtime information. Notice that AV2 dynamically detected all malware. However, this is expected because all the samples are a few years old, on average.

Table 1. Static (S) and Dynamic (D) tests for the original samples

Sample	AV1		AV2		AV3		AV4	
	S	D	S	D	S	D	S	D
S1	0	1	1	1	0	0	0	1
S2	0	0	1	1	0	0	0	1
S3	1	1	0	1	1	0a	0	0a
S4	1	1	1	1	1	0a	0	1

We evaluated the antiviruses by submitting jobs to our system. Each job described the original, the transformation, and the antiviruses to test. These jobs, given in Table 2, show that we found thousands of new malware variants.

Table 2. Jobs showing new HTML variants generated from originals

Sample	Transformation	Generated	Functional
S1	Permute all	24	24
S1	Subset all	42	20
S2	Permute all	101674	1293
S2	Subset all	48	0
S2	Subset node 27	32768	2
S2	Subset node 20	4096	8
S3	Permute node 30	720	66
S3	Subset node 30	64	2
S3	Subset node 20	4096	8
S3	Subset node 32	524288	1
S3’	Permute node 30	108053	30
S4	Permute all	193070	7493
S4	Subset all	81494	0
S4	Subset node 10	3526	0
S4	Subset node 5	60	0

Not all jobs ran to completion, which is why some of them generated fewer variants than others under the same transformation. We suspended generator threads when we saw that others were creating functional variants. This reduced the workload on the functional workers. The transformation *Permute all* means that each node has a permute transformation, and similarly for *Subset all*. Other transformations are per node, usually assigned to Javascript nodes of the abstract syntax tree. Since they are assigned to different parts of the tree, there are no overlapping variants. The sample S3' is essentially the same as S3, but has a different AST, hence node 30 does not correspond to node 30 from S3.

Table 3. Number of Statically (S) and Dynamically (D) undetectable malware variants

Sample	Transformation	Functional	AV1		AV2		AV3		AV4	
			S	D	S	D	S	D	S	D
S1	Permute all	24	24	24	0	0	24	24	24	0
S1	Subset all	20	20	20	0	0	20	20	20	12
S2	Permute all	1293	1293	1293	1293	0	1293	1293	1293	0
S2	Subset all	0	0	0	0	0	0	0	0	0
S2	Subset node 27	2	2	2	2	0	2	2	2	0
S2	Subset node 20	8	8	8	8	0	8	8	8	0
S3	Permute node 30	66	65	65	66	0	66	66	66	0
S3	Subset node 30	2	1	1	2	0	2	2	2	0
S3	Subset node 20	8	0	0	8	0	8	8	8	0
S3	Subset node 32	1	0	0	1	0	1	1	1	0
S3'	Permute node 30	30	0	0	30	0	30	30	30	2
S4	Permute all	7493	0	0	0	0	0	5818	7493	0
S4	Subset all	0	0	0	0	0	0	0	0	0
S4	Subset node 10	0	0	0	0	0	0	0	0	0
S4	Subset node 5	0	0	0	0	0	0	0	0	0

The detection effectiveness of the antiviruses is given in Table 3. The table shows that most of the functional variants that we have created are undetectable by at least one antivirus. It proves that antiviruses cannot be evaluated based on mutations only. It also indicates that any detection mechanism that assumes that mutations must preserve the semantics of the original [10], may fail to work.

Table 3 provides the raw results, unabridged. The last three rows are all zero because these jobs did not produce functional variants, and thus have been suspended. No sample was dynamically undetectable by AV2. However, manual experiments with this antivirus showed that a *map* transformation yields statically and dynamically undetectable malware variants. The map is not an obfuscation as it replaces HTML elements with non equivalent ones. Other variants can be obtained by composing two obfuscation methods (string partitioning

and BASE64 encoding). We interpret these experiments as evidence that our software is basic, and needs more transformations.

6 Conclusions

We have demonstrated that semantics-preserving obfuscation is not the only way to produce malware mutations. By relaxing the notion of obfuscation to that of semantically non-preserving transformations, we were able to obtain transformations that produce functional mutations. We developed a virtualized testing environment which allowed us, using two simple transformations, to generate thousands of samples which were undetectable by commercial AV products. Our results demonstrate the viability of obfuscation techniques that do not preserve code semantics. This does not mean that such mutations are more malicious, or should be considered as a replacement for traditional code obfuscation. Rather, they should be viewed as another threat, alone or in combination with other techniques, that anti-malware technology must be able to prevent.

References

1. jsoup: Java HTML parser. http://jsoup.org/apidocs/
2. Rhino. http://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
3. Abrams, R., Ghimiri, D., Smith, J.: Corporate AV/EPP comparative analysis - exploit evasion defenses. Technical report, NSS Labs (2013)
4. CERT UK. An introduction to malware (2014). www.cert.gov.uk/resources/best-practices/an-introduction-to-malware/
5. Christodorescu, M., Jha, S.: Testing malware detectors. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA 2004, ACM (2004)
6. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report 148, University of Auckland, New Zealand, July 1997
7. Filiol, E., Jacob, G., Liard, M.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. J. Comput. Virol. **3**(1), 23–37 (2007)
8. Granville, K.: 9 Recent Cyberattacks Against Big Business. New York Times, 5 February 2015
9. Kandias, M., Gritzalis, D.: Metasploit the penetration tester's guide. Comput. Secur. **32**, 268–269 (2013)
10. Kwon, J., Lee, H.: Bingraph: discovering mutant malware using hierarchical semantic signatures. In: MALWARE 2012, pp. 104–111. IEEE Computer Society (2012)
11. Maggi, F., Valdi, A., Zanero, S.: Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors. In: Enck, W., Felt, A.P., Asokan, N. (eds.) SPSM@CCS 2013, pp. 49–54. ACM (2013)
12. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.K.: A semantics-based approach to malware detection. ACM Trans. Program. Lang. Syst. **30**(5), 25 (2008)
13. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Chen, K., Xie, Q., Qiu, W., Li, N., Tzeng, W.-G. (eds.) ASIACCS 2013, pp. 329–334. ACM (2013)
14. Rastogi, V., Chen, Y., Jiang, X.: Catch me if you can: evaluating android anti-malware against transformation attacks. IEEE Trans. Inf. Forensics Secur. **9**(1), 99–108 (2014)

15. Xu, W., Zhang, F., Zhu, S.: In: MALWARE 2012, pp. 9–16. IEEE Computer Society (2012)
16. Zheng, M., Lee, P.P.C., Lui, J.C.S.: ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 82–101. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37300-8_5