Weaknesses in Defenses Against Web-Borne Malware
(Extended Abstract)

Gen Lu and Saumya Debray

Department of Computer Science
The University of Arizona, Tucson, AZ 85721, USA
{genlu, debray}@cs.arizona.edu

Abstract. Web-based mechanisms, often mediated by malicious JavaScript code, play an important role in malware delivery today, making defenses against web-borne malware crucial for system security. This paper explores weaknesses in existing approaches to the detection of malicious JavaScript code. These approaches generally fall into two categories: lightweight techniques focusing on syntactic features such as string obfuscation and dynamic code generation; and heavier-weight approaches that look for deeper semantic characteristics such as the presence of shellcode-like strings or execution of exploit code. We show that each of these approaches has its weaknesses, and that state-of-the-art detectors using these techniques can be defeated using cloaking techniques that combine emulation with dynamic anti-analysis checks. Our goal is to promote a discussion in the research community focusing on robust defensive techniques rather than ad-hoc solutions.

1 Introduction

The growing importance of the Internet in recent years has been accompanied by a corresponding increase in web-based malware delivery, e.g., via "drive-by downloads" [11, 12, 10]. Such attacks are often carried out via scripts written in JavaScript, a language commonly used in client-side web applications.

Thwarting such attacks requires the ability to detect malicious JavaScript code. However, this is not easy: attackers generally use a variety of techniques, such as dynamic code generation and server-side polymorphism, to create code that is highly obfuscated and inscrutable. Existing techniques for detecting malicious JavaScript, discussed in Section 2, typically focus on handling current obfuscation techniques. A natural question to ask, therefore, is: what are the weaknesses of current detection techniques, what sorts of cloaking techniques might malware use to exploit those weaknesses, and what might tomorrow's malware look like? This paper explores this question by analysing existing detection techniques for JavaScript malware to examine their assumptions and study how these assumptions can be broken. Further, as a proof-of-concept, we present a combination of obfuscation and anti-analysis techniques, targeting static and dynamic approaches respectively, against state-of-the-art detectors. Our experiments show that these techniques are effective in circumventing existing detection techniques.

2 JavaScript Malware

Howard catalogs various obfuscation techniques currently used by JavaScript malware to avoid detection [6]. In JavaScript, several methods are provided for executing a string dynamically, for example, eval() and document.write(). This process of introducing new code at runtime is called *code unfolding*. JavaScript malware found in the wild often adopt a combination of the techniques discussed above, with multiple levels of code unfolding and redirection, which makes it difficult to determine its intent from a static examination of the program text. It should be noted, however, that code obfuscation is also used for legitimate purposes, e.g., intellectual property protection and code compression. Obfuscation is therefore not, in itself, an indicator of malicious code.

Several authors have discussed the use of machine-learning-based classifiers trained to recognize malicious code [2–4, 14]. These approaches are generally lightweight and so are suitable for online or large-scale detection. A drawback of such approaches is that the classifiers learn to recognize current obfuscation techniques but have difficulty handling code that does not resemble current obfuscated malware. Additionally, purely-static approaches cannot handle obfuscations involving dynamic code generation via unfolding.

To address the issues arising from dynamic code unfolding, some researchers have proposed using execution monitoring, typically in a sandboxed environment, to observe runtime behaviors [3, 14]. Different approaches usually focus on different aspects of execution, such as memory objects, suspicious function invocations and sequence of actions. Some researchers have also proposed using static and/or dynamic techniques for detecting shellcode-like strings [5, 13, 18]. While dynamic analysis makes it possible to examine any code that may be created as the program executes, it usually suffers significant execution overheads resulting from monitoring and limited code coverage. Various multi-path exploration techniques also have been proposed to increase code coverage of above detection techniques [1, 9, 7]. Some recent proposals are lightweight enough to be practical for online analysis on a large scale [7].

3 Thwarting Analysis

This section considers how the limitations of existing detection techniques for JavaScript malware can be exploited to allow malicious code to evade detection. As the discussion in the previous section suggests, obfuscations aimed at evading existing detectors should satisfy three properties. First, the obfuscated code should look, at least syntactically, like ordinary unobfuscated JavaScript code. Second, the malware should avoid exposing its malicious behaviors if its execution is being monitored. Finally, to thwart multi-path exploration, it should avoid using conditional jumps to implement the control flow logic that activates the malicious code if no execution monitoring is detected.

One way to accomplish these goals is using a code obfuscation technique called emulation-based obfuscation [15, 16] together with anti-analysis defenses and a technique we call *implicit conditionals*. These techniques are not specific

to JavaScript, and emulation and anti-analysis techniques have been encountered in native-code malware. However, what makes them especially relevant to web-delivered malware is a combination of circumstances. First, the routine use of browsers, together with the proliferation of resource-limited devices such as smartphones, means that malware detection has to be cheap, lightweight, and online (i.e., has to occur as web pages or documents are opened for viewing). This requirement, combined with the increased code complexity resulting from technologies such as HTML5, limits the computational effort detectors can devote to code analysis. The remainder of this section explores how this observation can be exploited by constructing obfuscations that allow for a high degree of code diversity and require significant computational effort to penetrate, thereby rendering them likely to be able to escape detection.

3.1 Emulation-Based Obfuscation

Emulation-based obfuscation transforms the original JavaScript program P into a pair (B_P, I_P), where B_P is a bytecode representation of P and I_P is an interpreter written in JavaScript whose sole purpose is to execute the program B_P. While we do not know of existing JavaScript malware using this approach to obfuscation, the idea itself is not new to security researchers and similar techniques have already been adopted by native malware writers.

From an attacker's perspective, emulation-based obfuscation offers the advantage that the payload logic is not exposed: examining the executed code only reveals the structure and logic of the bytecode interpreter; the underlying logic of the program being executed is encoded in the form of bytecode as data. Moreover, details of the bytecode encoding and corresponding interpreter can be perturbed randomly, which means successful reverse engineering of one obfuscated program may not give us much help for analyzing programs obfuscated by the same obfuscator. Finally, emulation-based obfuscation has the significant advantage that the code looks syntactically similar to ordinary unobfuscated JavaScript code, making it harder to detect using machine-learning approaches.

In addition to concealing the logic of the malicious code, emulation can also be used to hide other components of the program, such as shellcode strings, that detectors often look for. This can be done by applying existing string obfuscation techniques to the shellcode strings, but instead of implementing the string decoding routine in JavaScript directly (which itself is suspicious and can be identified by existing detectors), transforming the decoding logic into bytecode as well. This makes it possible to conceal both the shellcode strings and the decoder from a static examination of the program.

3.2 Anti-Analysis Defense

Anti-analysis defenses, which are also encountered in native malware, involve detecting runtime monitoring/tracing system; if the program determines that its execution is being monitored, it can then alter its execution to avoid revealing any malicious behavior.

Ideally, a detection system should be indistinguishable from a true victim. This very often does not hold true in practice, however, because dynamic analyses are typically performed within sandboxed environments, which are susceptible to detection. One reason is that complete behavior emulation of web browser, including DOM, ActiveX controls and various plug-ins, can be quite difficult. Also, sandboxed detectors incur significant execution overhead. Our experiments indicate, for example, that sandboxed execution monitoring systems for JavaScript are 1–2 orders of magnitude slower than modern browsers. This dramatic difference in overheads between monitored and non-monitored execution environments suggests that measurements of execution speed may be used to detect runtime execution monitoring. We note, however, that timing tests to detect monitoring are not infallible, and sometimes there may not be a clear line between fast monitors and slow clients. This means that anti-analysis defenses evade detection at the possible cost of reduced exploitation success rate. On the other hand, overhead variation due to different browsers is usually not a problem, since each exploit typically targets vulnerability in a web-browser of specific version and/or brand.

3.3 Implicit Conditionals

It may be possible to bypass the anti-analysis defenses described in the previous section by combining dynamic analyses with multi-path exploration techniques [1, 9, 7]. Existing multi-path exploration techniques focus on conditional branches in the code: whereas a vanilla program execution will take any one branch of a conditional branch, multi-path exploration involves exploring both branches. From an analysis perspective, conditional branches have the advantage that straightforward code inspection allows us to determine, for any given conditional branch, the expression that is evaluated and the code addresses where execution continues depending on whether the branch is taken or not.

We can make multi-path exploration more difficult by replacing conditional branches with calculation of parameters used by the interpreter (discussed in Section 3.1) in a way that makes the selection of execution paths transparent. We refer to this approach as *implicit conditionals*. The intuitive idea here is that given an explicit conditional $C \equiv$ **if** e **then** C_0, we replace C by a code fragment C' that has the following properties:

1. C' does not contain an explicit test on e.
2. If e holds, the effect of executing C' is identical to that of executing C_0; otherwise, executing C' has no or meaningless effect.

Since the execution of C' is not predicated on e, it is executed in all cases, but this is set up in such a way that the parameters used by the interpreter (i.e. instruction-pointer, entry-point, etc.) have the correct values if and only if e holds. This can be done in various different ways using a function f_e that satisfies the following properties: (i) f_e computes some appropriate desired value if and only if the condition e holds; and (ii) the computation of f_e does not involve conditionals. We list below some ways of using such conditional-free functions.

Entry point generation. The idea here is that the initial value of the interpreter's instruction pointer, i.e., the offset in the byte-code array where the execution of the byte code program begins, is determined by a conditional-free function f_e that takes as input an environment profile (i.e., a collection of values describing the program's execution environment) and returns the correct value only if the condition e holds. This can be done in many different ways; here we present an example based on the anti-analysis defense discussed in Section 3.2. In this case, the environment profile p is the time required to execute some given fragment of code. Suppose that we have determined that the p should be less than 100 (ms) in target browser, and the bytecode offset of the entry point for the malicious code is $entry_m = 20$, then f_e might be implemented as:

$$f_e(p) = \lceil \frac{p+1}{100} \rceil \times 20$$

In this case, $f_e(p) \equiv entry_m$ (i.e. 20) if and only if $p \in [0, 99]$, which ensures the attack runs normally; for $p \geq 100$, $f_e(p) \geq 40$, and the execution ends up with unpredictable behavior.

However, unpredictable behavior may not be guaranteed to be non-suspicious. For example, even if the value returned by f_e is not the correct value $entry_m$, it may nevertheless expose some components of an attack, e.g., a heap spray or construction of a shellcode string, that can cause the attack to be recognized, or the program might crash, which itself can be considered suspicious. One way to deal with this using a more elaborate computation for the function f_e such that, if the condition e does not hold, returns a value that is out of bounds in the bytecode array. Or a better approach is to construct bytecode sequence deliberately, such that, while only the correct value leads to malicious behavior, all the other entry-point values calculated from possible inputs are corresponding to valid and harmless bytecode execution without crash.

Figure 1 shows an example of applying entry point generation for implicit conditional. Detailed discussion of Figure 1 will be presented in Section 3.4.

Instruction pointer increment generation. In this case, the amount by which the interpreter's instruction pointer is incremented after each instruction is determined by a conditional-free function that returns the correct value only if e holds. Typically, the instructions of (non-branching) bytecode are laid out contiguously in memory and the instruction pointer is incremented by the size of a single instruction each time an instruction is executed. Such contiguous layout is not essential, however: for example, each real instruction can be separated by one or more "chaff instructions" such that proper execution requires that the instruction pointer be incremented by some multiple of the size of a single instruction. The value of this increment can then be set using an implicit conditional, similarly to the entry point generation described above. More generally, the amount by which the instruction pointer is incremented after each instruction need not be a constant: for example, it can be a sequence of pseudo-random numbers, each in some range $[min, max]$: all we need is a predictable sequence of values such that bytecode instructions can be placed at the correct offsets. The

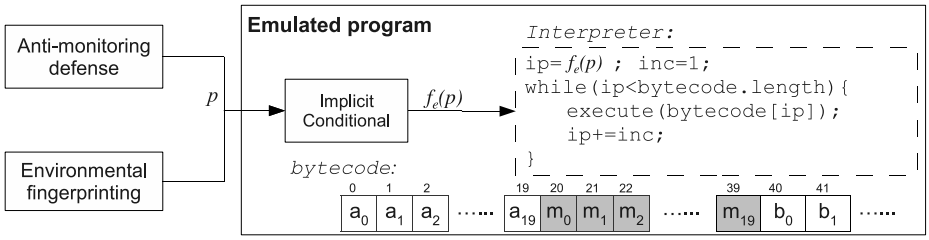

Fig. 1. General structure of a program combining emulated-obfuscation, anti-analysis defense and implicit conditionals implemented by entry point generation.

function f_e can then be used to set the seed for the pseudo-random sequence to the right value if and only if e holds.

3.4 Implementation

Figure 1 shows the general structure of a program combining all the proposed techniques discussed in this paper, namely, emulated-obfuscation, anti-analysis defense and implicit conditionals. As we can see from the high-level structure shown in Figure 1, anti-analysis defense, alone with other environmental fingerprinting code are located at the beginning of the program. Their result – environment profile p is then passed to the implicit conditional. In this example, implicit conditional is implemented by entry point generation alone as discussed in Section 3.3, and the instruction pointer increment is 1. Furthermore, the conditional-free function $f_e(p)$ is designed to return 20 if and only if p shows the intended condition holds and returns $20 * i$ where $i >= 2$ and $i \in integer$ otherwise. $f_e(p)$ is then used to set the entry point of the bytecode program. Finally, the bytecode is arranged such that bytecode instructions $bytecode[20], bytecode[21], \ldots, bytecode[38], bytecode[39]$ (corresponding to dark slots in the array), when executed in this order, will lead to malicious behavior, execution starts with other possible entry points (e.g. $40, 60, 80, \ldots$) would cause the emulation to behave harmlessly (one simple way to implement this is to assign bytecode nop-slide in light-colored slots).

As the proof-of-concept implementation, we have applied all proposed anti-analysis techniques on existing malware and benign programs by manual transformation. For example, all the samples discussed in Section 4 are implemented by hand using an arbitrarily chosen, stack-based instruction set; and the anti-analysis defense and implicit conditionals are both implemented in their basic forms (i.e. single loop for anti-analysis defense, and simple instruction-pointer and entry-point generation as shown in Section 3.3).

4 Experimental Evaluation

To evaluate if the proposed techniques are effective against existing detectors, we selected 7 real malware samples, named M_1 to M_7, including 6 scripts in

Sample	File Type	CVE Number	OSVDB ID
M_1	HTML	-	64839
M_2	HTML	CVE-2006-3730	27110
M_3	HTML	-	80662
M_4	HTML	CVE-2007-3071	38803
M_5	HTML	CVE-2007-3703	37707
M_6	HTML	-	61964
M_7	PDF	CVE-2008-2992	49520

Table 1. Description of malware samples

Malware Sample	Existing Obfuscation			New Obfuscation		
	VirusTotal	Wepawet	Zozzle	VirusTotal	Wepawet	Zozzle
M_1	5 / 40	√	×	0 / 42	×	×
M_2	4 / 41	√	×	0 / 42	×	×
M_3	5 / 42	√	√	0 / 42	×	×
M_4	5 / 42	√	√	0 / 42	×	×
M_5	5 / 42	√	×	0 / 42	×	×
M_6	5 / 42	√	×	0 / 42	×	×
M_7	10 / 42	√	n/a	2 / 42	×	n/a

√: detected ×: undetected

Table 2. Detection Results of obfuscated malware samples from existing detectors. For fractions present in columns "VirusTotal", the denominator is the number of anti-virus software available on VirusTotal, and the numerator is the number of anti-virus software that identify corresponding sample as malicious.

HTML pages and one in a PDF file (see Table 1, where *OSVDB ID* is the identification number used by the open source vulnerability database [17]). All the samples use heap-spray for payload delivery. Next we created two sets of obfuscated programs from these. Programs in the first set had two different obfuscators applied to them, each of them using existing techniques such as string obfuscation and unfolding. Those in the second set were obfuscated using the techniques proposed here as described at the end of the previous section. It should be noted that applying proposed obfuscation doesn't affect the reliability of malware, which was tested by running obfuscated exploit in browser with targeted plugins installed.

We used three malware detectors, covering a wide spectrum of detection technologies, for our experiments: VirusTotal [19] is an online portal to a collection of anti-virus software with up-to-date exploit databases that exemplifies current commercial malware detection technology; Zozzle [4] is a machine learning based static detector (we used the same trained classifier as evaluated in [4] for our experiment); and Wepawet [20], a hybrid detection system based on JSAND [3], that represents a state-of-the-art combination of static and dynamic analyses. We believe these three detectors, range from traditional signature matching to state-of-the-art static and dynamic analyses, represent the current state of detection techniques. Therefore it allows us to have a comprehensive evaluation on the effectiveness of proposed obfuscation techniques against different approaches.

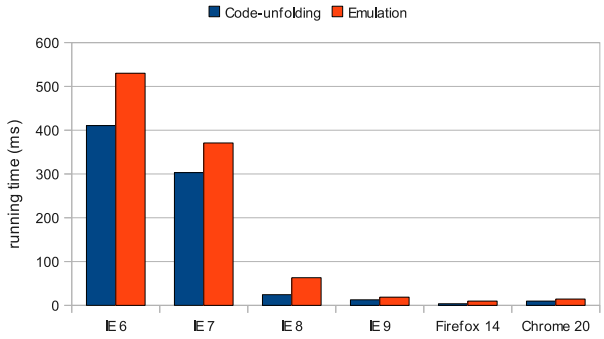

Fig. 2. Comparison of average running time between current obfuscation and emulation

Table 2 shows the detection rates for these three malware detectors. There is no result for neither version of M_7 from Zozzle, since Zozzle is designed for detecting web-based JavaScript malware. It can be seen that, while the malware samples obfuscated by existing techniques were identified as malicious with 100% detection rates by both VirusTotal and Wepawet, and with 33% detection rate by Zozzle, another group of malware samples, protected by new obfuscation techniques, were able to bypass all the targeting detectors, with the exception of M_7, which was detected by two of the anti-virus software on VirusTotal. It turns out, however, that this has nothing to do with any malicious content: the only reason the PDF file M_7 is identified as malicious is that it contains JavaScript code. We confirmed this with a PDF file containing just a Fibonacci number program written in JavaScript, which is identified as malicious by the same two anti-virus software with identical exploit names.

We also compared the relative performance of emulator-based obfuscation with current approaches to obfuscation using code unfolding on several different browsers, result is shown in Figure 2. We used the same testcases as discussed above, but with anti-analysis defense removed from emulated samples (since it is not limited to emulated code). We can see that the running times for the two different obfuscation techniques are comparable, even for older browsers. The samples obfuscated using emulation are slightly slower than code-unfolding based samples, but the differences are not very large. This suggests that such obfuscation techniques could realistically be deployed using current technology.

5 Discussion

Space constraints preclude a detailed discussion of attack models against the propose approach; the interested reader is referred to the full version of this paper [8]. Here we give a brief sketch of the difficulties an adversary would have in attacking this scheme.

The key point to note is that the proposed scheme does not have to be impossible to break—all that is needed for our approach to be effective is that the analyses necessary to break it should be sufficiently expensive that they are unsuitable for routine use in online detectors. As observed earlier, the use of

emulation allows us to avoid string-based obfuscation of program text, resulting in ordinary-looking JavaScript code. The structure of the interpreter structure can be masked by merging additional nodes and edges into the control flow graph of the interpreter so as to camouflage its structure. Data used in exploits, e.g., shellcode strings, can be kept in encoded form and decoded at runtime. Together, these imply that simple static analysis will not be enough to distinguish malicious emulated code from other ordinary JavaScript code; rather, dynamic analysis will be necessary. However, such dynamic checks necessarily incur an additional cost, which can be detected via anti-analysis checks, allowing the program to take an execution path that does not expose any malicious content. Finally, the analysis of alternative execution paths is made more difficult by using implicit conditionals.

While these obstacles against detection can all be overcome, we believe that the analyses powerful enough to do this will necessarily incur enough computational cost as to be impractical for routine use in online detectors.

6 Conclusion

In recent years, malware delivered through infected web pages has become an important delivery mechanism for malware. Very often this is done using JavaScript code, making the detection of malicious JavaScript code an important problem. Current proposals in the research literature for detecting JavaScript malware, although proved to be effective, are closely tied to existing malware and obfuscations. In this paper we discuss the limitations of existing detection techniques and describe ways in which such detectors can be evaded. Experiments show that the proposed techniques can hide existing JavaScript malware from state-of-the-art detectors. Our goal is not to suggest that this particular approach to obfuscation is the only possible—or even the most important, effective, or likely—way around current defenses; rather, it is to show current ad-hoc detection methods can be easily defeated, and promote a deeper discussion in the research community about the assumptions underlying current detection techniques and possible approaches for defending future attacks regardless of obfuscation.

Acknowledgments

We are grateful to Giovanni Vigna for providing access to the Wepawet system for our experiments, and to Ben Zorn, Ben Livshits and Timon Van Overveldt for their help in evaluating our work using Zozzle. Nathan Yee helped with the collection and testing of malicious code. This work was supported in part by the National Science Foundation (NSF) via grant nos. CNS-1016058 and CNS-1115829, and the Air Force Office of Scientific Research (AFOSR) via grant no. FA9550-11-1-0191. The opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR or NSF.

References

1. David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. 2008.
2. D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proc. 20th International Conference on World Wide Web*, pages 197–206, 2011.
3. M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. 19th International Conference on World Wide Web*, pages 281–290, 2010.
4. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser JavaScript malware detection. In *USENIX Security Symposium*, 2011.
5. M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proc. 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.
6. F. Howard. Malware with your mocha: Obfuscation and anti emulation tricks in malicious JavaScript, September 2010. http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf.
7. Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.
8. Gen Lu and Saumya Debray. Weaknesses in defenses against web-borne malware. Technical report, Dept. of Computer Science, The University of Arizona, February 2013. `http://www.cs.arizona.edu/~debray/Publications/js-emulobf.pdf`.
9. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.
10. N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proc. 17th USENIX Security Symposium*, pages 1–15, 2008.
11. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, 2007.
12. N. Provos, M.A. Rajab, and P. Mavrommatis. Cybercrime 2.0: when the cloud turns dark. *Communications of the ACM*, 52(4):42–47, 2009.
13. P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, pages 169–186. USENIX Association, 2009.
14. Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.
15. VMProtect Software. Vmprotect software protection, 2008. http://vmpsoft.com/.
16. Oreans Technologies. Themida: Advanced windows software protection system, September 2008. http://www.oreans.com/themida.php.
17. The open source vulnerability database. http://http://www.osvdb.org/.
18. Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E.P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.
19. Virustotal. https://www.virustotal.com/.
20. Wepawet. http://wepawet.cs.ucsb.edu.