HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs

Aurore Fass, Michael Backes, and Ben Stock
CISPA Helmholtz Center for Information Security: {aurore.fass, backes, stock}@cispa.saarland

***Abstract*—In the malware field, learning-based systems are getting popular to detect new malicious variants. Nevertheless, it has been shown that attackers with *specific* and *internal* knowledge of a target system may be able to produce input samples which will be misclassified. In practice, the assumption of *strong attackers* with insider information is not realistic. Therefore, we present HideNoSeek, a novel and generic camouflage attack, which evades the entire class of detectors based on syntactic features, without needing any information about the system it is trying to evade. Our attack consists of changing the constructs of a malicious JavaScript sample to imitate a benign syntax. In particular, HideNoSeek uses malicious seeds and searches for similarities at the Abstract Syntax Tree (AST) level between the seeds and traditional benign scripts. Thereby, benign sub-ASTs are replaced by identical malicious ones, and the benign data dependencies are adjusted–without changing the AST–, so that the malicious semantic is kept after execution. In practice, we are able to generate 51,853 malicious scripts from 21 malicious seeds and 8,279 benign web pages. In addition, we can hide on average eight malicious samples in a benign AST of the Alexa Top 10, and nine among five of the most popular JavaScript libraries.**

I. Introduction

JavaScript is a browser scripting language initially created to enhance the interactivity of websites and to improve their user-friendliness. However, as it offloads the work to the user's browser, it is also used to engage in malicious activities such as crypto mining, drive-by download attacks, or redirections to websites hosting malicious software [10, 33, 47]. Given the prevalence of such nefarious scripts, the anti-virus industry has increased the focus on their detection [15, 28, 41, 45, 64]. The attackers, in turn, make increasing use of obfuscation techniques, e.g., string manipulation, dynamic arrays, encoding obfuscation [74], to evade detection by traditional anti-virus signatures and to impose additional hurdles to manual analysis. Yet, using the way in which lexical (e.g., keywords, identifiers) or syntactic (e.g., statements, expressions) units are arranged in a given JavaScript file provides valuable insight to capture the salient properties of the code. This way, several systems leveraged the combination of lexical or syntactic features with machine learning, to automatically and accurately detect new malicious (obfuscated) variants [16, 19, 50, 60].

Nevertheless, the field of attacks against systems using machine learning for classification purpose is vast [4, 5]. In particular, several attacks have already been proposed in, e.g., the image or the malware fields to evade classifiers by transforming a given input sample so that it keeps its intrinsic properties, but the classifier's predictions between the original and the modified input differ [17, 23, 25, 27, 39, 51, 52, 58, 62, 69, 76]. For them to work, all these tools need information about the classifier they are trying to evade, like some knowledge about the target model internals, or the training dataset, or at least the classification scores assigned to input samples. Another class of attacks focusses on the transferability in machine learning. Indeed, adversarial examples affecting one model often affect another, even if they have different architectures or training sets, provided they were trained to perform the same task. Therefore attackers can build and train their surrogate classifier, craft adversarial examples against it and transfer them to the victim classifier [42, 56, 57, 67, 70]. Still, the attackers need a specific target system, as well as access to it, for them to train their surrogate classifier.

In this paper, we introduce a novel class of attacks which works independently of any machine learning system and does therefore not need any knowledge of model internals, training dataset, or a classifier to test. Indeed, HideNoSeek leverages the fact that malicious obfuscation techniques leave traces in the syntax of the malicious files, which enables to differentiate them from benign (even obfuscated) inputs. As a consequence, changing the constructs of a malicious sample to imitate a benign syntax by design foils any approach that leverages the syntactic or lexical structure for classification. In particular, HideNoSeek automatically replaces sub-ASTs (Abstract Syntax Trees) of a benign JavaScript input with a malicious file's AST, while retaining the malicious semantics in the modified "benign" file. As a consequence, any classifier working on the syntactic structure is *by construction* bypassed by our approach. Since a variety of benign samples and libraries can be chosen for malicious replacement, our attack is also effective against AV-systems using structural analysis, e.g., signatures or content-matching.

Our implemented attack responds to the following challenges: practical applicability, effectiveness, and high impact. We address these challenges by proposing a methodology to detect, replace and adjust so-called clones at the AST level between a benign and a malicious file. The key elements of HideNoSeek are the following:

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345).

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-60-6
https://dx.doi.org/10.14722/madweb.2019.23003
www.ndss-symposium.org

- *Program Dependency Graph-Based Analysis* — Our system benefits from a syntactic analysis to transform JavaScript code into an AST. The latter is used to build a Control Flow Graph (CFG), which is leveraged to define a Program Dependency Graph (PDG), representing–besides the flow of control–the data dependencies between the nodes.

- *Slicing-Based Clone Detection* — The previous PDG is then combined with backward slicing to detect syntactic clones– i.e., isomorphic subgraphs between a benign and a malicious file–, with respect to control and data flows.

- *Benign AST Replacement* — Once found, the benign clones are replaced by malicious ones. The remaining benign code is then automatically semantically modified so that it a) is still able to run, and b) keeps its original AST-based structure, which will, therefore, foil syntactic detectors.

- *Comprehensive Evaluation* — We evaluate our system in terms of the proportion, validity, and complexity of the malicious samples it crafts, as well as the impact these documents would have. For the malicious seeds, we use 23 syntactically unique (deobfuscated) files extracted after thorough analysis and clustering of our 122,345 sample set. As for the benign samples, we consider the scripts extracted from the start pages of the Alexa top 10,000 websites, as well as 268 widely used JavaScript library versions. Overall, HideNoSeek crafted 51,853 malicious files with a benign AST mimicking the exact syntax of scripts from Alexa Top 10k.

The remainder of this paper is organized as follows. Section II introduces state-of-the-art JavaScript obfuscation techniques and static detection systems. We describe the methodology and implementation of HideNoSeek in Section III. Subsequently, in Section IV we present the results of our evaluation w.r.t. to the quantity, quality and impact of the crafted samples; the implications of that evaluation are further discussed in Section V. Finally, Section VI presents related work and Section VII concludes.

II. JavaScript Obfuscation

This section first provides an overview of existing JavaScript obfuscation techniques. Then, we select state-of-the-art static systems, which can detect malicious (obfuscated) JavaScript. Finally, we introduce HideNoSeek, our advanced method to hide malicious inputs into benign ASTs.

A. Obfuscation Techniques

To avoid detection by traditional AV-malware detectors, attackers abuse obfuscation techniques. Several categories of evasion can be found in the wild [32, 40, 74]:

- *Randomization obfuscation* consists of randomly inserting or changing elements of a script without altering its semantics, e.g., whitespace characters or comments addition, variables name randomization, which foils techniques relying on content matching.
- *Data obfuscation* regroups different string manipulation techniques, such as the combination of string splitting and string concatenation, or character substitution.
- *Encoding obfuscation* avoids that a given string appears in plaintext by using standard, e.g., ASCII, or custom encoding, as well as encryption and decryption functions.

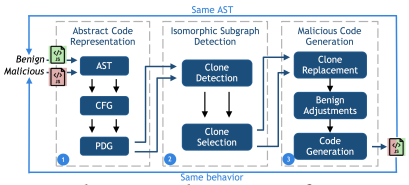


Figure 1: Schematic depiction of our approach

- *Logic structure obfuscation* consists of adding irrelevant instructions, which can take the form of numerous conditional branches, in the target script.

Still, obfuscation should not be confused with maliciousness: benign obfuscation can protect intellectual property, while malicious obfuscation hides the malicious intent of the sample. Therefore, benign, malicious, or no obfuscation leave different traces in the syntax of the considered files, which can be leveraged for an accurate malware detection.

B. Static Detection Systems

Several systems combine the previous differences at a lexical, syntactic, or structural level with off-the-shelf supervised machine learning tools to distinguish benign from malicious JavaScript inputs. Due to their usage of static features, these systems represent a subset of the detectors HideNoSeek targets. In particular, Rieck et al. developed Cujo [60], which combines an n-gram analysis of JavaScript lexical units with an SVM classifier for an accurate detection of malicious inputs. Similarly, Stock et al. presented Kizzle [64], which uses tokens extracted from different exploit kits families for clustering and signature generation. Moreover, Curtsinger et al. implemented Zozzle [16], which combines the extraction of features from the AST, as well as the corresponding JavaScript text, with a Bayesian classification system to identify syntax elements highly predictive of malware. Hao et al. also used a naive Bayes classification algorithm [28] to analyze JavaScript code by benefitting from extended API symbol features through the AST. With JaSt, Fass et al. [19] leveraged the use of syntactic units, combined with a random forest classifier, to accurately detect new obfuscated JavaScript instances. Still, these systems do not confound obfuscation with maliciousness (c.f. Section II-A), but leverage specific constructs for an accurate detection.

C. Malicious Transformation of ASTs

Instead of trying to hide the maliciousness of a file behind obfuscation layers, which are specific to malware and thereby enable their detection, HideNoSeek changes the constructs of a malicious sample to imitate a benign syntax. As a consequence, it automatically foils the previously outlined classifiers. The main idea is to rewrite a malicious AST into an existing benign one. To this end, it first looks for similarities between the malicious and the benign ASTs. Since malicious obfuscation is responsible for their differences, the first step consists of deobfuscating the malicious file, to get the original syntax which resembles more a benign AST than the obfuscated version. JSDetox [66] and box-js [12] are combined with a manual analysis for the deobfuscation.

III. Methodology

The architecture of our system consists of an abstract code representation part, a clone detector, as well as a malicious

```
1 var x = 1;
2 var y = 1;
3 if (x == 1) {d = y;}
```

Listing 1: JavaScript code example

code generator, as shown in Figure 1. First, we perform a static analysis of JavaScript documents, augmenting the traditional AST with control and data flow information, which are stored in a PDG. HideNoSeek then uses the new graph structure to detect isomorphic subgraphs, which we refer to as *clones*, between a benign and a malicious input. Finally, the benign clones are replaced by the malicious ones, and the modified file is automatically adjusted with respect to the AST so that its malicious semantics is still executed. In the following sections, we discuss the details of each stage.

A. Program Dependency Graph Analysis

In order to detect clones at the AST level between a benign and a malicious file, with respect to control and data flows, HideNoSeek is based on an abstract, labeled and oriented code representation. The AST provides a hierarchical decomposition of the source file into syntactic elements, as well as code abstraction, ignoring, for example, the variable names and values to consider them as *Identifier* or *Literal* (note that for legibility reasons, the variable names and values actually appear in the graphical representations this paper contains, but they are not part of the graphs). The control and data flow between the graph's nodes are indicated by labels on the AST, which becomes a PDG.

1) Syntactic Analysis: The syntactic analysis is performed by the open-source JavaScript parser Esprima [29], which takes a valid JavaScript sample as input and produces an ordered tree (the AST) describing the syntactic structure of the program. Overall the Esprima parser can produce up to 69 different syntactic units, referred to as nodes. Inner nodes represent *operators* such as *VariableDeclaration*, *AssignmentExpression* or *IfStatement*, while the leaf nodes are *operands*, e.g. *Identifier* or *Literal* (with the exception of a *ContinueStatement* or a *BreakStatement*). As an illustration, Figure 2a shows the Esprima AST obtained from the code snippet of Listing 1. As presented in the graph, the AST only retains information about how the programming constructs are nested to form the source code, but does not contain any semantic information such as the control or data flow, which we need for clone detection.

2) Control Flow Analysis: Contrary to the AST, the CFG allows to reason about the interplay of statements, in particular, the order in which they are executed, as well as the conditions that have to be met for a specific execution path to be taken. To this end, statements (predicates and non-predicates) are represented by nodes that are connected by labeled and directed edges to represent flows of control.

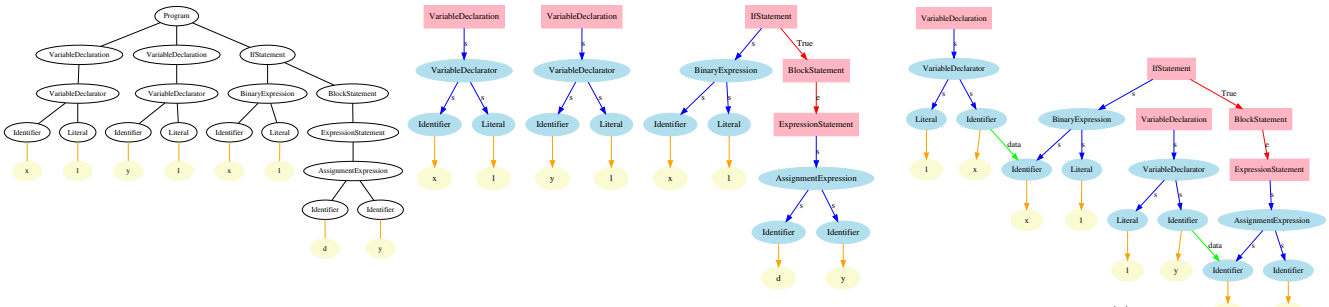
We construct the CFG by traversing the previous AST's nodes depth-first pre-order. Since the Esprima AST does not only comprise statements, but also contains non-statement and still non-terminal information, as shown in Figure 2a, we first define a *statement dependency* edge (labeled with *s*). It consists of linking a statement node, characterized as such by the JavaScript grammar [18], to its non-statement children, and recursively link them to their own non-statement descendants all the way down to the leaves. After, we define two different labels for the CFG edges linking two statement nodes. The label *e* is used for edges originating from non-predicate statements, while edges originating from predicates are labeled with a boolean, standing for the value the predicate has to evaluate to, for this path in the graph to be chosen, as shown in Figure 2b. Contrary to the AST of Figure 2a, this graph shows an execution path difference when the *if* condition is true, and when it is not. Nevertheless, the CFG still does not contain any data flow information, which we also need for clone detection.

3) Data Flow Analysis: To this end, we implement a PDG [22], which augments the previous CFG with data dependencies. For this purpose, statements are connected by a directed data dependency edge if and only if, an element, e.g., variable, object, function, defined or modified at the source node is used at the destination node, taking into account the reaching definitions for each variable, as shown in Figure 2c. This PDG indicates, in particular, the order in which statements from Listing 1 should be executed, e.g., as suggested by the data flows, lines 1 and 2 are executed before line 3; the lines 1 and 2 could nevertheless be interchanged without altering the code semantics.

In JavaScript, a scope defines the accessibility of variables. If a variable is defined outside of any function, or without the *var*, *let* or *const* keywords, or using the *window* object, it is in the global scope, whereas variables that can be used only in a specific part of the code, e.g., block statement, are considered to be in a local scope. To build our PDG, we traverse the previous CFG depth-first pre-order and thereby maintain two variables lists. The former contains the global variables, and the latter the local variables currently declared in the *considered block statement*, taking into account the fact that variables defined with the *let* or *const* keyword have their own local scope in the block where they were defined. As far as objects are concerned, we keep the order in which they are modified, since we cannot statically predict which method should be called on it first, e.g., an *XMLHttpRequest* must be opened before the *send()* method is called. As a consequence, we consider the data dependencies on the *complete* object and that an object is modified whenever a method is called on it, or one of its property changed. In these cases, we implement a data dependency between the previous object version and the current one and update our variables list (local or global according to the context) with a reference to the modified object. As far as functions are concerned, we handle their name as a variable (local or global), since functions and variables cannot share a name in JavaScript. In particular, we make the distinction between function declarations–a standalone construct defining named function variables–, and function expressions–named or anonymous functions that are part of larger expressions. Furthermore, HideNoSeek respects the function scoping rules, and handles closures and lexical scoping. Finally, the function call nodes are connected to the corresponding function definition nodes with a data dependency, thus defining the PDG at the program level [77].

B. Slicing-Based Clone Detection

Given the space $\mathbb{B}$ of benign JavaScript samples and the space $\mathbb{M}$ of malicious ones (according to some oracle), we aim



(a) (b) (c)

Figure 2: AST (a), CFG (b) and PDG (c) corresponding to the code of Listing 1

at building a sample space S so that:

$$S = \{x | x \in \mathbb{M}, \exists x' \in \mathbb{B} | ast(x) = ast(x')\}$$

with $ast(x)$ the AST of the sample x.

This can be achieved by first detecting malicious sub-ASTs from a given file that can also be found in a benign document, with respect to control and data flows. We refer to such common structures as clones. To this end, the previous representation of JavaScript code into a PDG, combined with the use of program slicing is used. The strongest clones are then selected according to metrics we defined.

1) Equivalence Classes: Finding clones between a benign and a malicious file consists of finding isomorphic subgraphs between the two documents. To this end, we consider the algorithm of Komondoor et al. [46] which combines PDGs and program slicing [73] for this purpose. Because of slicing, this algorithm can find non-contiguous clones (i.e., clones whose components do not occur directly one after the other in the source code), as well as clones in which matching statements have been reordered. Indeed, the PDG provides a certain level of abstraction, e.g., it ignores the variables' name and value to consider them as *Identifier* or *Literal*. Even though they appear in the graphical representations this paper contains, they are not part of the graphs, and we added them to the figures only for legibility reasons. Furthermore, the PDG enables to bypass the arbitrary sequencing choices made by the programmer and instead captures only the dependencies (data and control flows) between the different program components, which justifies our decision for this code representation.

First, HideNoSeek partitions the benign PDG statement nodes into equivalence classes based on their syntactic structure. For example, the PDG of Figure 2c would have four distinct classes: *VariableDeclaration* (containing two elements), *IfStatement*, *BlockStatement*, and *ExpressionStatement*. Then, the previous equivalence classes are completed with the considered malicious file, e.g., the PDG of Figure 3 would add two malicious elements in the class *ExpressionStatement*. At this stage, it is not sure that a benign and a malicious node from the same class match, as they could have a different subgraph, which is the case in the previous example. We do this test in the next step.

```
1 wscript = WScript.CreateObject('WScript.Shell');
2 wscript.run("cmd.exe /c \"<malicious powershell>;\"", "0");
```

Listing 2: Malicious JavaScript code example

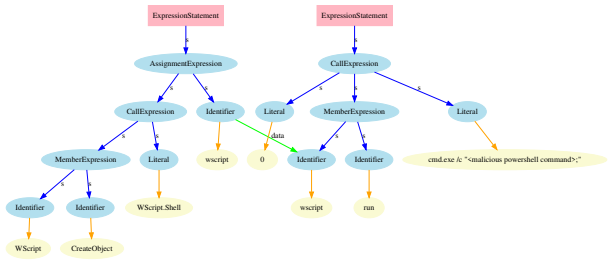


Figure 3: PDG corresponding to the code of Listing 2

2) Clone Detection: The next step consists of iterating through the previous equivalence list: for each equivalence class, the *find_clone* function, described in Algorithm 1, is called on every benign and malicious pair *(b, m)*. To find two isomorphic subgraphs, the former containing b and the latter m, HideNoSeek verifies that they have the same complete sub-AST by traversing and comparing their respective nodes along the *statement dependencies*. Then it slices backward in lock step along the data and control dependencies, starting from b and m, adding them as well as their predecessor to one slice if and only if their respective predecessor match (i.e., same class and same sub-AST). We iterate the process as long as predecessors that have not been handled yet are found. In addition, whenever we find a pair of matching statement nodes that we have already handled, the process stops for the current pair, the system retrieves the clones which have been found previously on the pair and adds these nodes to the current slice. Besides performance improvement, it also ensures that no subsumed clones are reported at this stage. Furthermore, when a pair of non-matching statement nodes *(b, m)* is tested, the system still recursively slices backward from b and tests its predecessors against m, which enables to jump over benign data flow dependencies to find more non-continuous clones. Because of this step, we can found two isomorphic subgraphs which are not PDGs, therefore expanding the possible set of clones. For example, HideNoSeek detects that the ASTs of Listing 2 and Listing 3 match respectively for the lines 2 and 3 (format: *a.b(str1, str2)*). By slicing backward along the data dependencies, our system respectively tests the lines 1 and 2, which do not match. Applying the previous rule, it respectively tests the lines 1 and 1 which match (format: *a = b.c(str)*). This way, the complete AST of Listing 2 can be found in Listing 3. To avoid infinite loops on each pair tested, a list is kept to handle them only once. When the process finishes, it has identified two isomorphic subgraphs: one benign and one malicious, which may contain several nodes. Further pairs of isomorphic subgraphs (independent from the previous ones) may be found while iterating through

```
obj = document.createElement("object");
obj.setAttribute("id", this.internal.flash.id);
obj.setAttribute("type", "application/x-shockwave-flash");
obj.setAttribute("tabindex", "-1");
createParam(obj, "flashvars", flashVars);
```

Listing 3: Initial extract of the plugin jPlayer 2.9.2 (benign)

the equivalence classes, hence the need for some metrics to determine the strongest pair of clones.

Data: benign, malicious, clones_list
Result: clones_list entry with the corresponding benign and malicious subgraphs
initialization;
if *benign and malicious belong to the same equivalence class and have the exact same complete sub-AST* **then**
  if *they have already been handled together* **then**
    search the corresponding clones_list entry;
    append the clones found so far to it;
  else
    create a a new clones_list entry;
    add benign and malicious to it;
    $ben\_parents \leftarrow$ backward_slice(benign);
    $mal\_parents \leftarrow$ backward_slice(malicious);
    iterate over ben_parents and mal_parents;
    call find_clone on the resulting combinations;
  end
else
  $ben\_parents \leftarrow$ backward_slice(benign);
  iterate over ben_parents;
  call once find_clone(ben_parent, malicious);
end
return clones_list

Algorithm 1: *find_clone()*: finds two isomorphic subgraphs between a benign and a malicious PDG

3) Strongest Clones Selection: A portion of the same malicious AST can be found several times in the benign one (the opposite may also be true). In this case, HideNoSeek selects only one. The first criterion consists of choosing the largest clone (based on the number of matching statement nodes it contains), and not a subsumed version of it, so as to maximize the clone coverage, knowing that subsumed clones can only be reported when the system jumps over a non-matching benign node to consider its data flow predecessors. The second criterion consists of maximizing the proportion of identical tokens between the benign and the crafted samples. Indeed, mimicking the AST automatically copies most of the tokens, but we may observe some differences for the syntactic unit *Literal*, which can be translated into several tokens, e.g., *Int*, *Numeric*, *Null*, depending on the context. If some tokens do not match, HideNoSeek reports them and suggests how to modify them, for them to match the initial tokens, e.g., the *Bool false* is equivalent to the *String "0"*, and to the *Int 0*. The third and last criterion consists of minimizing the distance between the nodes inside a clone, therefore minimizing the adjustment surface (Section III-C2).

Nevertheless, HideNoSeek does not necessarily report clones for all *(b, m)* pairs tested, as they may have different syntactic structures. For this purpose, we semi-automatically generated different syntactic versions of a malicious file to improve the proportion of clones reported (c.f. Section IV-A1). For example, the *VariableDeclaration var a = 10* (in top-level code), the *AssignmentExpression a = 10*, and the *ExpressionStatement window.a = 10* are semantically equivalent, but syntactically different.

C. Malicious Code with a Benign AST

Once HideNoSeek finds one (or several) unique malicious AST equivalent in the benign file, it replaces the benign sub-AST with the malicious one. This process then yields some adjustments for the benign code to still be able to run.

1) Clone Replacement: An AST is composed of inner nodes and of leaf-nodes, the latter which represent the operands. Saying that a benign AST is identical to a malicious AST means that they have the same nodes, with the same oriented edges. Still, the benign code is different from the malicious one, which is possible as the variables name are not directly contained in the AST, but rather are attributes of the leaf nodes. As a consequence, replacing the attributes of the benign leaf nodes with the malicious ones would replace the benign code portion, previously selected by HideNoSeek, by the malicious code, while keeping the same AST structure. The lines 1 and 3 of Listing 4 illustrate the replacement of Listing 2 in the corresponding part of Listing 3 (c.f. Section III-B2). Nevertheless, the replacement process has modified the benign environment and might, therefore, interfere with the benign functionalities, which could result in the modified sample not running anymore.

2) Benign Adjustments and Code Generation: As a countermeasure, HideNoSeek searches for fragments that may have been impacted by the replacement process and automatically adjusts them to the environment, so that the modified code still runs. To this end, it recursively explores the data dependencies originating from benign clone nodes, under the conditions that (a) they do not belong to a cloned node and (b) they have not been handled yet, e.g., lines 2, 4 and 5 from Listing 3. HideNoSeek first renames the benign variables, impacted by the replacement, with the name of the malicious variables which are now part of the code. Then, it analyses the end of each data dependency, recursively storing the nodes it contains in a list, all the way down to the leaves. After that, our system searches in its database a sublist of the previous nodes list to determine the generic modifications that have to be done to the benign nodes, for the code to still run while keeping its initial AST structure (if HideNoSeek does not find a match in the database, it reports the missing pattern so that we can search for a new adjustment and add it to the database). For example, if *['CallExpression', 'Identifier']* matches the node list, it means that the benign code would look like *func(my_obj[params])*, where *func* and *params* are respectively a benign given function with its parameters, and *my_obj* stands for the object the data dependency points to, i.e. the object that HideNoSeek modified. As a consequence and because of our modification, the function may not run anymore. To avoid this phenomenon, HideNoSeek replaces the initial function name by a function which can be executed for every possible parameter type and number, without throwing an error or causing side effects. Such functions include, among others, *decodeURI()*, *isFinite()*, *toString()*. Our current list contains nine different names, randomly selected each time that such a replacement is needed. Line 5 of Listing 4 illustrates this specific adjustment process. Other adjustments may include a property or a method called on the object we modified, e.g., lines 2 and 4 of Listing 4. As previously, HideNoSeek has a list of nine properties, that can also be used as meth-

ods, such as *hasOwnProperty*, *toString*, *propertyIsEnumerable*, which can be combined and are valid in all contexts. Finally, the ECMAScript code generator Escodegen [65] is used to transform the modified AST back to JavaScript code.

```
wscript = WScript.CreateObject('WScript.Shell');
wscript.toString('id', this.internal.flash.id);
wscript.run('cmd.exe /c "<malicious powershell>;"', "0");
wscript.hasOwnProperty('tabindex', '-1');
parseFloat(wscript, 'flashvars', flashVars);
```

Listing 4: Modified extract of the plugin jPlayer 2.9.2 (Listing 3) with the malicious code of Listing 2

IV. Comprehensive Evaluation

In this section, we outline the results of our extensive evaluation. To produce malicious samples with a benign AST, HideNoSeek disposes of 23 unique malicious seeds, which can be hidden in a subset of our 8,546 different benign scripts. We first evaluated the number of malicious samples our system was able to produce per seed, before considering the impact our attack would have. Then, we verified the validity and maliciousness of the samples previously crafted. Finally, we tested HideNoSeek on real-world detectors and analyzed its run-time performance.

A. Experimental Setup

The experimental evaluation of our approach rests upon two extensive datasets. The former contains 122,345 unique (based on their SHA1 hash) malicious JavaScript samples, while the latter is comprised of 8,941 unique benign files.

1) Malicious Datasets: Our malicious dataset, presented in Table I, is a collection of samples collected between 2014 and 2018 (73% of which have been collected after 2017). In particular, it includes exploit kits provided by Kafeine DNC (DNC) [38] and GeeksOnSecurity (GoS) [24], as well as the malware collection of Hynek Petrak (Hynek) [59] and a local information security government agency (GA). We consider that all these files are malicious. Indeed, the deobfuscation and the manual analysis of these inputs, performed in the next step, enabled us to exclude the documents, which did not present any malicious behavior. The samples' deobfuscation was initially performed by JSDetox [66] and box-js [12], but could not be automated due to malicious files conducting environment detection and refusing to execute. As a consequence, each tested sample needed to be, at least *partially*, manually deobfuscated. For this purpose, we clustered our data (by source), based on the syntactic units it contained, using an n-gram analysis [15, 19, 50, 60]. From the 122,345 scripts, we got 61 clusters, which reduced the number of files to analyze manually. Subsequently, we randomly selected one file per cluster, deobfuscated and unpacked it until the initial payload appeared; i.e., to a stage were no JavaScript was dynamically created through means of *eval* or equivalents. In essence, this is the state we assume a malicious entity would have before obfuscation or packing. After deobfuscation, we noticed that eight samples were either benign or incomplete (e.g., we did not have the landing page of the exploit kit, which prevented us from unpacking the malicious content) and we could not find any valid substitute in the same clusters. In contrast, two files had two malicious behaviors depending on the machine where they were executed. Therefore, they gave us four deobfuscated samples instead of two.

Source	Creation	#JS	Clusters	Deobf
DNC	2014-18	4,444	9	11
Hynek	2015-17	30,247	15	15
GoS	2017	2,595	27	19
GA	2017-18	85,059	10	10
Sum	2014-18	122,345	61	55
VirusTotal	2017-18	13,884	8	8

Table I: JavaScript malicious dataset description

Source	#JS	#Valid JS
Alexa 10k	8,673	8,279
Libraries	268	267
Sum	8,941	8,546

Table II: JavaScript benign dataset description

Finally, we got 55 working malicious documents, 30 of which are droppers, 3 call a PowerShell command, 2 a VBScript command and 20 are exploit kits (e.g., donxref, meadgive, RIG).

To avoid duplicated samples, we manually iterated over the 55 scripts and looked for similar structures, e.g., the combination of *createElement* and *appendChild* is often semantically equivalent to *document.write*. As mentioned in Section III-B3, we kept the different variants found for HideNoSeek to test, in the case that it does not find a clone with the first one. Still, we refer to all these variants as *one* file. Besides, we are working at the AST level, therefore we consider here that two samples with the same AST but a different behavior are identical. After duplicate deletion, we retained 22 unique malicious seeds, to which we added a crypto-miner, as cryptojacking in browsers has recently become a widespread threat [31, 47, 72]. Finally, to verify to what extent our dataset was representative of the malicious distribution found in the wild, we extracted 13,884 additional samples from VirusTotal [68]. These samples were collected *after* the files we analyzed previously and did not contain any duplicate. As before, we clustered them syntactically and got 8 clusters (Table I), one file of each we deobfuscated. Since the 8 deobfuscated samples matched our 23-sample set (7 matched known exploit kits, and 1 a dropper), we deemed our dataset to be saturated.

2) Benign Datasets: As for the benign dataset (Table II), we statically scraped the start pages of Alexa top 10,000 websites, also including external scripts. Given the fact that this JavaScript was extracted from the start pages of high-profile sites, we assume them to be benign. At the same time, we downloaded the most popular JavaScript libraries according to W3Techs [71]; this information is leveraged to evaluate the impact our attack would have (Section IV-B2).

B. Evasive Samples Generation

HideNoSeek leverages the 23 malicious seeds to produce malware with a benign AST. In this section, we first report the number of samples that our system could craft per malicious seed, before evaluating the impact our attack would have on the highest ranked web pages and libraries.

1) Evasion per Malicious Seed: In our first experiment, we studied the number of samples that HideNoSeek could

produce per malicious seed, by using the Alexa top 10,000 web pages as a benign dataset. During the deobfuscation process (Section IV-A1), we noticed that exploit kits from the same family (based on AV labels) could have a different syntactic structure, as well as a different behavior. In these cases, they appear several times in the seeds from Table III. This table represents the number of malicious samples crafted per malicious seed (second and fifth columns), the number of nodes that HideNoSeek had to adjust due to the replacement of benign sub-ASTs with malicious ones (third and sixth columns), as well as the total number of nodes contained in the crafted samples (fourth and seventh columns). In particular, we make a distinction between the samples crafted with the benign AST of a top 1,000 web page, against a top 10,000 one. In fact, the number of crafted samples is not linear and, proportionally, we tend to produce more samples for the first 1,000 web pages (e.g., for Blackhole1 we could have expected to generate around 5,600 samples in Alexa top 10,000 web pages, but in practice we got 10% less; for Donxref2 we even got 40% less than expected). This phenomenon can rather be observed when a certain amount of clones has already been found on Alexa top 1,000 (from a threshold of around 100 samples); otherwise, the data is too sparse to be generalized. Still, the start pages of the 1,000 most consulted websites do not seem to be larger (in terms of delivered JavaScript) than the start pages of the top 10,000. It rather is the opposite since, on average, our PDGs contain more nodes for pages from Alexa top 10,000 than Alexa top 1,000. Nevertheless, the first 1,000 seem to have a more complicated structure with, in particular, more data dependencies: for each replacement HideNoSeek made, it had to adjust more nodes for the first 1,000 web pages. For this reason, we estimate that the higher complexity of the first 1,000 web pages was more favorable to hide malicious seeds, whose different statements highly depend on each other.

The success of our hiding process also depends on the syntactic structures the seeds contain, and to what extent their syntax can also be found in benign scripts. With the exploit kit Misc, HideNoSeek was able to generate an evasive sample for 78% of the pages from Alexa top 10,000. On the contrary, it was unable to craft samples for two malicious seeds, namely Dropper and RIG2. For both of them, the difficulty lay in the syntactic structures they used, which were never found in benign documents. For example, our dropper used three times the construct *new ActiveXObject("object")*, which we could, e.g., map to the benign construct *new RegExp("regexp")*; in our sample set, however, we found no such pattern. Therefore, we looked for a new syntactic construct, semantically equivalent to the previous one, but which could be found in benign documents too. For this purpose, we studied the most common structures between our malicious seeds and benign dataset. The following structure *a.b("")* was found in 105,463 statements that matched benign and malicious documents. As a consequence, we replaced the previous malicious dropper's construct with its equivalent *WScript.CreateObject("object")*, but we did not get any clone either. Nevertheless, our tool reported 360 crafted samples for the PowerShell seed (Table III), which is actually a dropper too. Therefore, an attacker could still hide a dropper in 360 web pages from Alexa top 10,000. As for RIG2, it contained complex syntactic structures, such as

	ALEXA-1k			ALEXA-10k		
Seed	#Samples	#Adjust	#Nodes	#Samples	#Adjust	#Nodes
Blackhole1	558	93	106,897	5,042	68	120,271
Blackhole2	558	52	106,897	5,040	37	120,179
Crimepack1	209	74	89,700	1,388	55	103,944
Crimepack2	76	41	112,308	806	52	118,599
Crimepack3	73	54	145,970	847	95	152,309
Crypto-miner	85	211	75,523	259	137	128,797
Donxref1	66	26	131,348	921	45	137,618
Donxref2	132	471	88,672	774	361	116,051
Dropper	0	-	-	0	-	-
EK	18	7	140,958	237	20	144,814
Fallout	5	65	175,568	61	32	168,924
Injected1	679	68	97,494	6,415	47	105,543
Injected2	366	155	111,244	3,205	66	124,678
Meadgive	431	58	109,712	4,253	45	121,024
Misc	683	27	96,434	6,487	6	104,459
Neclu1	95	86	77,799	380	96	103,028
Neclu2	450	59	110,201	4,405	70	122,074
Packer	42	56	111,339	428	77	140,209
PowerShell	22	16	117,451	360	23	144,478
RIG1	17	47	180,265	203	160	172,221
RIG2	0	-	-	0	-	-
VBScript1	584	15	100,254	5,276	8	114,526
VBScript2	552	48	105,613	5,066	16	118,566

Table III: Analysis of the proportion of samples crafted per malicious seed

Alexa top 10	#Samples	#Nodes
1 google.com	13	58,322
2 youtube.com	14	151,527
3 facebook.com	7	143,772
4 baidu.com	7	35,018
5 wikipedia.org	0	-
6 qq.com	7	54,450
7 yahoo.com	8	67,264
8 taobao.com	7	89,910
9 tmall.com	8	63,102
10 amazon.com	8	36,060

Table IV: Analysis of the number of samples that could be hidden in Alexa top 10

window.frames[0].document.body.innerHTML, that benign web pages might tend to simplify, e.g., by storing this statement into several variables. We highlight potential improvements for this process in Section V.

Overall, and out of the 23 malicious seeds HideNoSeek got as input, it was able to craft malware for 21 of them. In total, it produced 5,701 malicious samples with the benign AST of an Alexa top 1k web page, and 51,853 for the top 10k. We believe this number can be improved by using different syntactic structures for the seeds; as the most common patterns can be identified easily (as above), the malware authors could adjust the code to use those constructs.

2) Impact of the Attack: HideNoSeek is able to hide a given malicious seed into different web pages–while keeping their initial AST–, thereby static detectors would fail to see the maliciousness. As a second experiment, we studied the impact our attack would have by targeting specific domains. For that, we focused on hiding malicious JavaScript in the most frequented web pages and libraries. Table IV indicates how many malicious seeds HideNoSeek could hide in Alexa top 10 web pages. In particular, between 57% and 60% of our seeds could be hidden in the start pages of the two most visited web pages, which would maximize the impact, in terms of infected users through web page browsing, of our attack. Except for *wikipedia.org*, where no clones were reported, we could hide a third of our seeds in the other top 10 web pages. Still, we know that for the attack to be effective

in practice, the server of these pages would have to be compromised, so that the original web page could be replaced by our crafted one. Should that happen, our modified website version would be harder to spot than, e.g., the British Airways attack [44], because of its structure exactly mimicking a benign syntax. A second way of infecting pages consists of infecting the libraries that these websites use.

For our third experiment, we considered five of the most popular JavaScript libraries, based on the proportion of websites using them [71], and studied the number of malware we could hide inside (Table V). The proportion ranges from 13% to 56% and is a bit higher than from Alexa top 10, where on average 8 seeds could be hidden, compared to 9 in the libraries. Similar to Android malware in repackaged applications [9, 61, 79], we envision that HideNoSeek could alter benign libraries and present them as an improved version of the original one, for malicious purpose. More specifically, such a modification of jQuery 1.12.4 would affect 29.7% of the websites, according to [71].

C. Validity Tests

Based on the insights that HideNoSeek could leverage 21 out of 23 malicious seeds to craft 51,853 malicious scripts, which have the exact same AST as scripts extracted from start pages of Alexa top 10,000, in this section, we verify the validity and maliciousness of the produced samples.

1) Same AST for Crafted and Benign Scripts: First and by construction, all malicious samples crafted by HideNoSeek have the same AST as the benign scripts it used for the replacement processes. Without further testing, this guarantees that classifiers purely based on syntactic features (e.g. JaSt [19]) will not be able to distinguish them.

2) Same Tokens for Crafted and Benign Scripts: Second, most of the tokens are similar between an initial benign file and a crafted one. The minor differences may come from a *Literal node*, which can represent several tokens depending on the context (Section III-B3). On average, 0.15 token differ for each 51,853 file crafted from Alexa top 10,000 websites (containing on average 115,717 nodes). Depending on the detector our implementation is trying to evade, this may be sufficient to prevent the evasion; e.g., for Kizzle [64], our malicious sample would be clustered together with benign samples due to their choice of maximum distance within a cluster. Moreover, we would produce *at most* one such sample every ten crafted script, which we assume to be negligible when considering the impact our attack would have (e.g., jQuery is used by 73.5% of the websites and a malicious modification of the most widely used version 1.12.4 would affect 29.7% of these sites [71]).

3) Crafted Scripts Still Running: Third, HideNoSeek modified the syntactic structure of a benign input to hide a malicious script inside, which could result in the crafted sample not running anymore. To decrease the proportion of broken samples, we implemented a module, which is able to detect parts of the program that may have been impacted by our transformations–by following the data dependencies originating from our replacements (Section III-C2). Still, some adjustments may not be working in the specific context where they have been transplanted, e.g., trying to get the length of an *undefined* object will throw an error. In addition, HideNoSeek searches clones between a benign and a malicious input, with respect to control and data flows. Nevertheless, it still is valid to replace two independent benign sub-ASTs by two malicious sub-ASTs, with variables declared in the global scope, depending on one another. In this case, we have to ensure that the execution order of these two ASTs is respected to avoid *ReferenceError* at runtime. To verify the correct execution of our crafted samples, we used the library *jsdom* [36]–which emulates web browser functionalities, e.g., DOM elements–to test JavaScript implementations using web standards with *Node.js*. This is necessary to ensure that in our tests, we do not break functionality that requires DOM components. At the same time, however, this environment cannot be used to test scripts scraped from websites, as the JSDOM is essentially a placeholder, and the downloaded JavaScript often relied on specific constructs in the DOM. Therefore, we executed every crafted sample from standalone benign libraries, like jQuery, to verify that they could still run without throwing, e.g., a *ReferenceError*. Out of the 1,224 samples we crafted from jQuery, 846 were still able to run, which represents 69% of them. As stated in Section IV-C2, we rather consider the impact our attack would have by using the working crafted samples, than the proportion of working samples HideNoSeek generates. For this purpose, related work [17, 52, 70, 76] combined their implementation with an oracle, which dynamically tested the validity and maliciousness of the samples they produced. In our specific case, such an infrastructure could not be built due to the complexity of emulating environments specific to each web page that should have been tested. Still, we were able to use 23 malicious seeds to produce 846 working malicious versions of jQuery, which had the exact same AST as the original ones.

Libraries	Usage	Version	#Samples	#Nodes
jQuery	73.5%	1.12.4	13	35,511
Bootstrap	18.1%	3.3.7	10	10,973
Modernizr	11.4%	2.8.3	3	3,174
MooTools	2.4%	1.6.0	7	27,786
Angular	0.4%	1.7.5-min	11	60,234

Table V: Analysis of the number of samples that could be hidden among the most widely used JavaScript libraries [71]

4) Crafted Scripts With a Malicious Behavior: Last but not least, it is not sufficient to verify the executability of the samples; we also needed to ensure that the malicious parts were either called or could at least be triggered. For this purpose, we randomly selected two working crafted samples per malicious seed and executed them in a web browser. Thereby, we manually verified which malicious parts were already called if any, and which should still be called. In fact, the jQuery library defines objects and methods for future use and does therefore not necessarily directly call them. As a consequence, we searched all malicious parts in the considered scripts, triggered their execution whenever possible (e.g., a closure cannot be called) and verified their correct execution order. Out of the 28 samples we manually analyzed, 20 did present a malicious behavior.

D. Evaluation Against Real-World Systems

In this section, we classify the 51,853 samples–previously crafted by HideNoSeek– using several detectors and the five

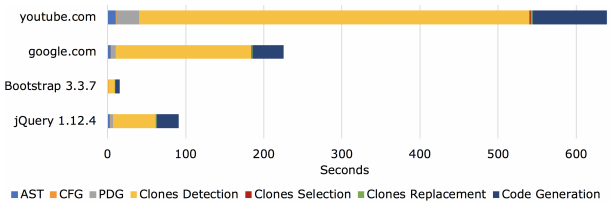


Figure 4: Time required to test our 23 malicious seeds on the two most popular websites and libraries

machine learning models presented in [19] (note that no scripts from Alexa were used to train the models, which are therefore independent of our test set and will not influence the classifiers' decisions). We averaged the detection results over five runs. By construction, our attack foils JaSt, which purely relies on the AST for malicious JavaScript detection. Still, it could detect 266.6 crafted samples (detection accuracy: 0.48%) [2]. Since these files have the same AST as the original benign documents, we have 266.6 false-positives in the benign seeds used to craft malware. On the contrary, Zozzle may include the text of the AST node as an additional feature, which could prevent the attack. Nevertheless, the system is not open-source, and we did not get any inputs from the authors. Based on the study of Cao et al. [11], we assume that the perfect mapping of benign ASTs, which thereby induces a lot of benign features in the malicious samples, would enable HideNoSeek to evade most of Zozzle's predictions. Last but not least, we followed the indication of Rieck et al. to reimplement the static part of Cujo. This system detected accurately 18.4 crafted samples (detection accuracy: 0.04%), 3.2 scripts of which changed classification between the benign and the malicious samples (6.2E-3%). Thereby, we assume that the tokens that might differ between the two file versions have a negligible impact, while considering the number of malware HideNoSeek crafts.

E. Run-Time Performance

The run-time performance of our system was tested on a commodity PC with a quad-core Intel(R) Core(TM) i3-2120 CPU at 3.30GHz and 8GB of RAM. The throughput evaluation was done on the two highest ranked Alexa web pages (*google.com* and *youtube.com*) and the two most popular JavaScript libraries (*jQuery* and *bootstrap*). Figure 4 presents the processing times, for all stages of HideNoSeek, to craft the 50 previous malware (Section IV-B2). The most time-consuming operation corresponds to the actual clones detection, which is NP-complete (Section III-B) and highly depends on the size of the PDGs (Table IV, Table V). The code generation phase is also relatively time-consuming as we traverse the PDGs of the crafted samples, so that Escodegen can produce the code back. Last but not least, the generation of the benign PDGs (each of them produced only once and stored for future use) may take some time depending on the size of the AST and the complexity of the code (number of data dependencies). Overall, the generation of the previous 50 samples took fifteen minutes.

V. Discussion

In this section, we first examine the limitations our attack might have, focusing on the *static* analysis of JavaScript. We then discuss existing defenses against attacks on machine learning systems and argue why they would not work for HideNoSeek. Still, we motivate some defenses that might prevent our system from crafting evasive samples. Finally, we introduce new strategies to improve our attack.

a) Limitations: HideNoSeek is based on a static analysis of JavaScript to build both the control and data dependencies in a given script. On the one hand, this approach provides a complete code coverage, evaluating all possible execution paths. On the other hand, it is subject to the traditional flaws induced by the high dynamic of the language [1, 21, 34, 35]. In particular, JavaScript can generate code at run-time, e.g., with the *eval* function, a dynamically constructed string can be interpreted as a program fragment and executed in the current scope. Still, HideNoSeek is resilient to many of these flaws, as it is applied to manually deobfuscated malicious samples. In particular, we specifically deleted all dynamic constructs to have the payload directly accessible (this should not be a problem to malware authors as they have the original malicious payload at their disposable).

b) Existing Defenses: As mentioned in Section I, the field of attacks against systems using machine learning for classification purpose, e.g., in the image or malware fields, is vast. Different studies assessed the security of learning-based detection techniques by evaluating the hardness of evasion, according to the information leaks an attacker might have, such as a black-box access to the classifier or dataset related inputs [8, 13, 14, 20, 70]. More recently, systems have been proposed to detect adversarial examples–i.e., inputs specifically crafted to foil a target classifier. They rely on the detection of unreliable results [63], statistical tests [26], dimensionality reduction [7, 75], the detection of adversarial perturbations [53, 54], or vectorization [37]. Nevertheless, we envision that none of them would work for our attack as we perfectly map an actual benign file syntactic structure.

c) Potential Detection Strategies: To exactly mimic a benign AST, the malicious seed first has to be deobfuscated, thus leaving its malicious logic in the open. Therefore, signatures might be able to detect our crafted samples. In practice, Virus Total analyzed the 28 samples selected in Section IV-C with between 42 and 57 different AV-systems. 4 crafted samples were detected (namely *Donxref1* and *PowerShell*, two times each), and by at most 2 AV-vendors. Even though the detection accuracy is very low, we envision that HideNoSeek could slightly obfuscate obvious malicious behavior, e.g., with percent-encoding, to completely bypass signature-detection. Another possibility to detect malicious inputs crafted by HideNoSeek would be to (a) recognize the original benign sample used for the hiding process, and (b) notice that it differs from the benign input it is supposed to be. In theory, if the original sample is recognized, a checksum test should indicate whether it is the original version or not. Nevertheless, the official library source code can also be altered for benign purposes, like functionality extensions or caching proxies, or stored together with other libraries. In particular, we used *retirejs* [55] to extract 73 different versions of jQuery used by Alexa top 10,000 web pages. Still, none of them matched the hash given on the official jQuery web page (for the previous reasons), essentially nullifying a checksum. Apart from this, HideNoSeek is an attack against *static* malicious JavaScript detectors, and does not necessarily

also foil hybrid or dynamic detectors such as Rozzle [45] or J-Force [43], which force the JavaScript execution engine to test all possible execution paths systematically. Similarly, Revolver [41] could detect that the original benign and the crafted sample have the same AST but that their classification results—according to its dynamic detector–differ, which would be labeled as an evasion attempt. Still, we provide a generic camouflage attack that evades the entire class of detectors based on syntactic features, also most of the lexical and structural detectors, without needing any information or access to the target systems.

d) Improving the Evasion: To improve the number of malicious samples HideNoSeek can generate, we envision that it could be paired with an intelligent syntactic obfuscator module. This system would be able to automatically transform a malicious syntactic structure into a semantically similar one, whose AST could be found in a benign file. We leave this implementation for future work.

VI. Related Work

HideNoSeek is a novel attack against malware detectors. Indeed, contrary to previously presented attacks, it does not need any information about the systems it evades. At the same time, it uses different data representations, e.g., AST and PDG, which are used in the fields of security analysis and clone detection too.

a) Adversarial Attacks: In the literature, several approaches have been proposed to evade targeted malware detectors, all of which need to have at least a black-box access to the system they are trying to evade. In particular, Šrndić et al. studied the range of possible attacks, according to the information leaks an attacker might have [70]. In addition, they explored the strategy of training a substitute model to find evading inputs, as well as the possibility to modify a malicious file so that it mimics the features of a chosen benign target [69]. Similarly, Fogla et al. introduced the polymorphic blending attacks to evade byte frequency-based network anomaly IDS by matching the statistics of the mutated attack instances to normal profiles [23]. Then, both Dang et al. and Xu et al. developed a system which stochastically manipulates malicious samples to find a variant, preserving the malicious behavior (oracle needed), while being classified as benign by the target (black-box access to the detector needed) [17, 76]. Contrary to the previous approaches, Maiorca et al. aimed at injecting malicious content in benign PDF documents so as to introduce minimum differences within its benign structure, while having a malicious behavior (reverse mimicry) [52]. Last but not least, Grosse et al. adapted the algorithm of Papernot et al. [58] to find which features should be changed to craft adversarial samples in the malware field [27].

b) PDG for Security Analysis: HideNoSeek can also be compared to systems using ASTs or PDGs for vulnerability detections. For example, LangFuzz from Holler et al. automatically crafts valid JavaScript samples based on inputs known to have caused invalid behavior before [30]. In particular, it replaces a given code fragment of an input file by a fragment of the same type (according to the grammar), while we replace a benign chunk by a syntactically equivalent malicious one (with respect to control and data flows in our case). Similarly, Yamaguchi et al. guided the search for new exploits by extrapolating known vulnerabilities using structural patterns extracted from the AST, which enabled them to find similar flaws in other projects [78]. To mine a more significant amount of source code for vulnerabilities, Yamaguchi et al. later introduced the code property graph–merging AST, CFG, and PDG into a joint data structure–to inspect the code structure with respect to control and data flows [77]. This new data structure was also used by Backes et al. to identify different types of Web application vulnerabilities [3].

c) PDG for Clone Detection: HideNoSeek also rests upon a clone detection algorithm to carefully spot isomorphic subgraphs between benign and malicious ASTs. First, Koschke et al. proposed a token-based clone detection algorithm based on suffix trees. Nevertheless, it yields clone candidates whose syntactic units might differ [48]. On the contrary, Baxter et al. introduced in 1998 an algorithm capable of detecting exact and near-miss clones over program fragments by means of ASTs [6]. Then, Krinke et al. considered PDGs, as an abstraction of the source code semantics, to identify similar code in programs [49]. Last but not least, Komondoor et al. combined PDGs with the use of program slicing to find clones in C programs [46]. The addition of the slicing part enabled them to find non-contiguous clones, clones in which matching statements have been reordered, as well as clones intertwined with each other.

VII. Conclusion

Many malicious JavaScript samples today are obfuscated to hinder the analysis and the creation of signatures. Nevertheless, these specific evasion techniques tend to leave recurrent traces in the syntax of malware, thereby contributing to their detection by lexical or syntactic classifiers. In this paper, we proposed HideNoSeek, a generic camouflage attack, which evades the entire class of syntactic detectors, as well as most of the lexical and structural systems, without needing any information about (or access to) the target systems. In fact, it changes the constructs of malicious samples to imitate benign syntax. The key elements of our approach are the following: (a) a modeling of the control and data flows extracted from the malicious seeds to hide, and from the benign files providing their AST as hiding place; (b) a detection and analysis of isomorphic sub-ASTs, with respect to control and data dependencies, between the previous benign and malicious inputs; (c) the replacement and adjustment of benign sub-ASTs by their malicious equivalents and (d) the evaluation of HideNoSeek on an extensive dataset of both malware and benign scripts. In practice, our approach is highly effective with its production of 51,853 malware from 21 malicious seeds and 8,279 benign web pages. In addition, it has a high impact: on average HideNoSeek crafts 8 malicious samples in each Alexa top 10 web page, and 9 in the five JavaScript libraries among the most commonly used. This way, we envision that our attack could alter benign libraries and present them as an improved version of the original one, for malicious purpose. In particular, we could hide 13 malicious seeds in jQuery 1.12.4, which would affect 27.7% of all websites [71].

References

[1] E. Andreasen and A. Møller, "Determinacy in Static Analysis for jQuery," in *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[2] Aurore54F, "JaSt - JS AST-Based Analysis," In: https://github.com/Aurore54F/JaSt. Accessed on 2018-12-17.

[3] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and Flexible Discovery of PHP Application Vulnerabilities," in *S&P*, 2017.

[4] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The Security of Machine Learning," *Machine Learning*, 2010.

[5] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can Machine Learning Be Secure?" in *ASIACCS*, 2006.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *International Conference on Software Maintenance (ICSM)*, 1998.

[7] A. N. Bhagoji, D. Cullina, C. Sitawarin, and P. Mittal, "Enhancing Robustness of Machine Learning Systems via Data Transformations," *Annual Conference on Information Sciences and Systems (CISS)*, 2018.

[8] B. Biggio, G. Fumera, and F. Roli, "Multiple Classifier Systems for Adversarial Classification Tasks," in *International Workshop on Multiple Classifier Systems*, 2009.

[9] J. Boutet, "Malicious Android Applications: Risks and Exploitation - A Spyware story about Android Application and Reverse Engineering," In: https://www.sans.org/reading-room/whitepapers/threats/malicious-android-applications-risks-exploitation-33578. Accessed on 2018-09-14.

[10] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages," in *International Conference on World Wide Web (WWW)*, 2011.

[11] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "JShield: Towards Real-time and Vulnerability-based Detection of Polluted Drive-by Download Attacks," in *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[12] CapacitorSet, "box-js - A tool for studying JavaScript malware," In: https://github.com/CapacitorSet/box-js. Accessed on 2018-05-28.

[13] N. Carlini and D. Wagner, "Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods," in *ACM Workshop on Artificial Intelligence and Security*, 2017.

[14] ——, "Towards Evaluating the Robustness of Neural Networks," *arXiv preprint arXiv:1608.04644v2*, 2017.

[15] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code," in *International Conference on World Wide Web (WWW)*, 2010.

[16] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and Precise In-Browser JavaScript Malware Detection," in *USENIX Security*, 2011.

[17] H. Dang, Y. Huang, and E.-C. Chang, "Evading Classifiers by Morphing in the Dark," in *CCS*, 2017.

[18] Ecma International, "ECMAScript 2018 Language Specification (ECMA-262, 9th edition, June 2018)," In: https://www.ecma-international.org/ecma-262/9.0. Accessed on 2018-06-27.

[19] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript," in *DIMVA*, 2018.

[20] A. Fawzi, O. Fawzi, and P. Frossard, "Analysis of Classifiers' Robustness to Adversarial Perturbations," *Machine Learning*, 2015.

[21] A. Feldthaus and A. Møller, "Semi-Automatic Rename Refactoring for JavaScript," in *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987.

[23] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic Blending Attacks," in *USENIX Security*, 2006.

[24] GeeksOnSecurity, "Malicious Javascript Dataset," In: https://github.com/geeksonsecurity/js-malicious-dataset. Accessed on 2018-07-13.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[26] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, "On the (Statistical) Detection of Adversarial Examples," *arXiv preprint arXiv:1702.06280v2*, 2017.

[27] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial Perturbations Against Deep Neural Networks for Malware Classification," in *European Symposium on Research in Computer Security*, 2017.

[28] Y. Hao, H. Liang, D. Zhang, Q. Zhao, and B. Cui, "JavaScript Malicious Codes Analysis Based on Naive Bayes Classification," in *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2014.

[29] A. Hidayat, "ECMAScript Parsing Infrastructure for Multipurpose Analysis," In: http://esprima.org. Accessed on 2018-09-16.

[30] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *USENIX Security*, 2012.

[31] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How You Get Shot in the Back: A Systematical Study About Cryptojacking in the Real World," in *CCS*, 2018.

[32] F. Howard, "Malware with your Mocha? Obfuscation and anti emulation tricks in malicious JavaScript," In: https://www.sophos.com/en-us/medialibrary/pdfs/technical%20papers/malware_with_your_mocha.pdf. Accessed on 2018-08-09.

[33] L. Invernizzi, S. Benvenuti, M. Cova, P. M. Comparetti, C. Kruegel, and G. Vigna, "EvilSeed: A Guided Approach to Finding Malicious Web Pages," in *S&P*, 2012.

[34] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the Eval That Men Do," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[35] S. H. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *International Symposium on Static Analysis (SAS)*, 2009.

[36] jsdom, "jsdom - A JavaScript implementation of the WHATWG DOM and HTML standards, for use with node.js," In: https://github.com/jsdom/jsdom. Accessed

on 2018-11-12.
[37] V. M. Kabilan, B. Morris, and A. Nguyen, "VectorDefense: Vectorization as a Defense to Adversarial Examples," *arXiv preprint arXiv:1804.08529v1*, 2018.
[38] Kafeine, "MDNC - Malware don't need coffee," In: https://malware.dontneedcoffee.com. Accessed on 2018-09-27.
[39] A. Kantchelian, J. D. Tygar, and A. D. Joseph, "Evasion and Hardening of Tree Ensemble Classifiers," in *International Conference on Machine Learning*, 2016.
[40] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger, ""NoFus: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"," in *Microsoft Research Technical Report*, 2011.
[41] A. Kapravelos, Y. Shoshitaishvili, M. Cova, and C. Krügel and Giovanni Vigna, "Revolver: An Automated Approach to the Detection of Evasive Web-based Malware," in *USENIX Security*, 2013.
[42] Z. Khorshidpour, S. Hashemi, and A. Hamzeh, "Evaluation of Random Forest Classifier in Security Domain," *Applied Intelligence*, 2017.
[43] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-Force: Forced Execution on JavaScript," in *WWW*, 2017.
[44] Y. Klijnsma, "Inside the Magecart Breach of British Airways: How 22 Lines of Code Claimed 380,000 Victims," In: https://www.riskiq.com/blog/labs/magecart-british-airways-breach. Accessed on 2018-09-14.
[45] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking Internet Malware," in *S&P*, 2012.
[46] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *International Symposium on Static Analysis (SAS)*, 2001.
[47] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense," in *CSS*, 2018.
[48] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in *Working Conference on Reverse Engineering*, 2006.
[49] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Working Conference on Reverse Engineering (WCRE)*, 2001.
[50] P. Laskov and N. Šrndić, "Static Detection of Malicious JavaScript-Bearing PDF Documents," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.
[51] D. Lowd and C. Meek, "Adversarial Learning," in *International Conference on Knowledge Discovery in Data Mining (KDD)*, 2005.
[52] D. Maiorca, I. Corona, and G. Giacinto, "Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection," in *ASIACCS*, 2013.
[53] D. Meng and H. Chen, "MagNet: A Two-Pronged Defense Against Adversarial Examples," in *CCS*, 2017.
[54] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On Detecting Adversarial Perturbations," in *International Conference on Learning Representation (ICLR)*, 2017.
[55] E. Oftedal, "Retire.js: What you require you must also retire," In: https://retirejs.github.io/retire.js/. Accessed on 2018-10-31.
[56] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples," *arXiv preprint arXiv:1605.07277*, 2016.
[57] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical Black-Box Attacks Against Machine Learning," in *ASIACCS*, 2017.
[58] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings," in *Euro S&P*, 2016.
[59] H. Petrak, "Javascript Malware Collection," In: https://github.com/HynekPetrak/javascript-malware-collection. Accessed on 2018-07-17.
[60] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks," in *Annual Computer Security Applications Conference (ACSAC)*, 2010.
[61] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications," in *Annual Computer Security Applications Conference (ACSAC)*, 2014.
[62] C. Smutz and A. Stavrou, "Malicious PDF Detection using Metadata and Structural Features," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.
[63] ——, "When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors," in *NDSS*, 2016.
[64] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A Signature Compiler for Detecting Exploit Kits," in *Dependable Systems and Networks (DSN)*, 2016.
[65] Y. Suzuki, "ECMAScript Code Generator," In: https://github.com/estools/escodegen. Accessed on 2018-06-15.
[66] Sven, "JSDetox - A Javascript malware analysis tool using static analysis / deobfuscation techniques and an execution engine featuring HTML DOM emulation," In: http://www.relentless-coding.org/projects/jsdetox. Accessed on 2018-05-24.
[67] F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "The Space of Transferable Adversarial Examples," *arXiv preprint arXiv:1704.03453v2*, 2017.
[68] VirusTotal, "VirusTotal - Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community," In: https://www.virustotal.com. Accessed on 2018-10-04.
[69] N. Šrndić and P. Laskov, "Detection of Malicious PDF Files Based on Hierarchical Document Structure," in *NDSS*, 2013.
[70] N. Šrndic and P. Laskov, "Practical Evasion of a Learning-Based Classifier: A Case Study," in *S&P*, 2014.
[71] W3Techs, "Usage of JavaScript libraries for websites," In: https://w3techs.com/technologies/overview/javascript_library/all. Accessed on 2018-11-13.
[72] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks," in *European Symposium on Research in Computer Security (ESORICS)*, 2018.
[73] M. Weiser, "Program Slicing," in *International Conference on Software Engineering (ICSE)*, 1981.
[74] W. Xu, F. Zhang, and S. Zhu, "The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study," in *International Conference on Malicious and Unwanted Software (MALWARE)*, 2012.

[75] W. Xu, D. Evans, and Y. Qi, "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks," in *NDSS*, 2018.
[76] W. Xu, Y. Qi, and D. Evans, "Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers," in *NDSS*, 2016.
[77] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *S&P*, 2014.
[78] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized Vulnerability Extrapolation Using Abstract Syntax Trees," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.
[79] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces," in *ACM Conference on Data and Application Security and Privacy*, 2012.