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 have become popular to detect new malicious variants. Nevertheless, attackers with *specific* and *internal* knowledge of a target system may be able to produce input samples which are misclassified. In practice, the assumption of *strong attackers* is not realistic as it implies access to insider information. We instead propose 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 malicious JavaScript samples to reproduce a benign syntax. For this purpose, we automatically rewrite the Abstract Syntax Trees (ASTs) of malicious JavaScript inputs into existing benign ones. In particular, HideNoSeek uses malicious seeds and searches for isomorphic subgraphs between the seeds and traditional benign scripts. Specifically, it replaces benign sub-ASTs by their malicious equivalents (same syntactic structure) and adjusts the benign data dependencies–without changing the AST–, so that the malicious semantics is kept. In practice, we leveraged 23 malicious seeds to generate 91,020 malicious scripts, which perfectly reproduce ASTs of Alexa top 10,000 web pages. Also, we can produce on average 14 different malicious samples with the same AST as each Alexa top 10. Overall, a standard trained classifier has 99.98% false negatives with HideNoSeek inputs, while a classifier trained on such samples has over 88.74% false positives, rendering the targeted static detectors unreliable.

CCS CONCEPTS

• **Security and privacy** → *Web application security*; *Malware and its mitigation.*

KEYWORDS

Web Security, Malicious JavaScript, Adversarial Attacks, AST

ACM Reference Format:
Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3345656

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3345656

1 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 [46], drive-by download attacks, or redirections to websites hosting malicious software [11, 32]. Given the prevalence of such nefarious scripts, the anti-virus industry has increased the focus on their detection [15, 27, 40, 44, 69]. The attackers, in turn, make increasing use of obfuscation techniques [81], e.g., string manipulation, dynamic arrays, encoding obfuscation, to evade detection by traditional AV-signatures and impose additional hurdles to manual analysis. Yet, using the way in which JavaScript's lexical (e.g., keywords, identifiers) or syntactic (e.g., statements, expressions) units are arranged provides valuable insight to capture the salient properties of the code. When combined with machine learning, such systems can automatically and accurately detect new malicious (obfuscated) variants [16, 18, 49, 64]. While such static analyses allow to quickly discard benign samples, forwarding only those likely to be malicious to more costly dynamic components [11], they are also heavily dependent on reliable lexical or syntactic detectors.

Therefore, the field of attacks against machine learning systems is vast [4, 5]. In particular, several attacks have been proposed 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, e.g., adversarial attacks on images [24, 61], on malware [22, 26, 38, 50, 51, 66, 74], and mutations of malicious samples [17, 80]. 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 focuses 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 [41, 59, 60, 72, 75]. Still, the attackers need a specific target system, as well as access to it, for them to train their own classifier.

In this paper, we introduce a novel attack which works independently of any machine learning system and does not need any knowledge of model internals, training dataset, or a classifier to test. Indeed, HideNoSeek leverages the fact that malicious obfuscation leaves traces in the malicious files' syntax, which enables to differentiate them from benign (even obfuscated) inputs. Thus, changing the constructs of a malicious sample to reproduce an existing benign syntax by design foils any classifier based on the syntactic or lexical structure. Due to the exact mapping onto a benign AST, our attack is more effective than existing malware, which are, e.g., inserted in bigger benign files, to evade detection by statistically increasing

the proportion of benign features. In particular, HideNoSeek automatically rewrites the AST of a malicious JavaScript input into an existing benign one, while retaining the malicious semantics, thereby bypassing any classifier working on the syntactic structure. Since we can choose a variety of benign samples' and libraries' ASTs to reproduce, our attack is also effective against AV-systems using structural analysis, e.g., signatures or content-matching. In essence, HideNoSeek specifically crafts samples that are likely to be labeled as benign by static pre-filtering systems, meaning they will not be analyzed by dynamic components.

Our implemented attack responds to the following challenges: practical applicability regarding crafted samples, effective evasion, and high impact in terms of misclassifications. We address these challenges by proposing a methodology to detect, replace, and adjust so-called clones at the AST level between benign and malicious files. The key elements of HideNoSeek are the following:

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

- *Slicing-Based Clone Detection* — Using backward slicing with respect to the control and data flow, we traverse a benign and malicious AST to detect isomorphic subgraphs (syntactic clones).

- *Benign AST Replacement* — Once found, the benign clones are replaced by the malicious ones. The crafted code is then automatically adjusted, with respect to the AST, to still be able to run.

We evaluate our system in terms of the proportion, validity, and complexity of the malicious samples it crafts, the impact these documents would have, as well as their evasion capability in practice. For the malicious seeds, we use 23 syntactically unique (deobfuscated) files extracted after a thorough analysis 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 popular JavaScript library versions. Overall, HideNoSeek crafted 91,020 malicious files which perfectly reproduce ASTs of Alexa top 10k, and evaded the targeted classifiers over 88.74% of the time.

The remainder of this paper is organized as follows. Section 2 introduces state-of-the-art JavaScript obfuscation techniques and static detection systems. We describe the methodology and implementation of HideNoSeek in Section 3. Subsequently, in Section 4 we present the results of our evaluation w.r.t. to the quantity, quality, impact and effective evasion of the crafted samples; the implications of that evaluation are further discussed in Section 5. Finally, Section 6 presents related work and Section 7 concludes the paper.

2 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 rewrite the ASTs of malicious inputs into existing benign ones.

2.1 Obfuscation Techniques

To avoid detection by traditional AV-malware detectors, attackers abuse obfuscation techniques. Several categories of evasion can be found in the wild, as stated by [31, 39, 81]:

- *Randomization obfuscation* consists of randomly inserting or changing elements of a script without altering its semantics, e.g., whitespace characters addition, variables name randomization, which foils techniques relying on content matching.
- *Data obfuscation* regroups string manipulation techniques, e.g., string splitting/concatenation, 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.
- *Logic structure obfuscation* consists of adding irrelevant instructions, e.g., numerous conditional branches, to the script.
- *Environment interactions* is specific to Web JavaScript, where statements can be split and scattered across multiple *script* tags in the HTML document. This way, the payload can be stored within the DOM and extracted subsequently.

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.

2.2 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, they represent a subset of the detectors HideNoSeek targets. Such static systems are particularly relevant to quickly analyze a considerable amount of files and forward only those likely to be malicious to much slower dynamic components [11]. In particular, Rieck et al. developed Cujo [64], which combines a lexical analysis of JavaScript with an SVM classifier for an accurate malware detection. Similarly, Stock et al. presented Kizzle [69], 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 predictive of malware. Hao et al. also used a naive Bayes classification algorithm [27] to analyze JavaScript code by benefitting from extended API symbol features through the AST. With JaSt, Fass et al. [18] leveraged the use of syntactic units, combined with a random forest classifier, to accurately detect new malicious (obfuscated) JavaScript instances. Still, these systems do not confound obfuscation with maliciousness (c.f. Section 2.1), but leverage specific constructs for an accurate detection.

2.3 Malicious Transformation of ASTs

Instead of trying to hide the maliciousness of a file behind traditional obfuscation layers, which are specific to malware and thereby enable their detection, HideNoSeek changes the constructs of a malicious sample to reproduce an existing benign syntax (this hiding

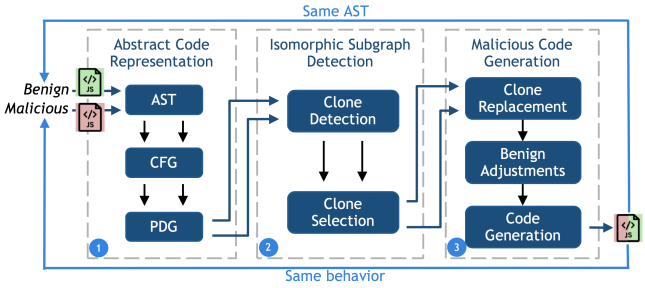

Figure 1: Schematic depiction of our approach

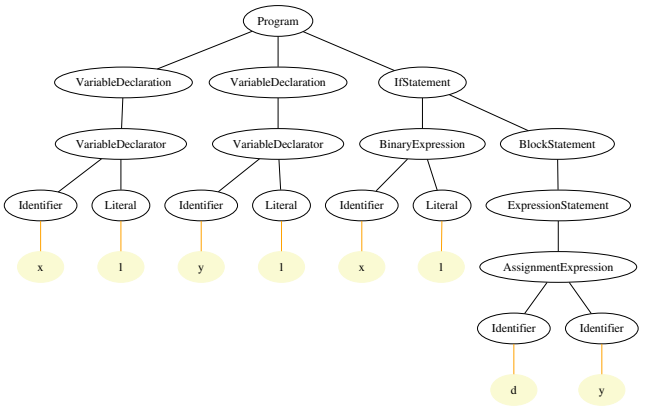

Figure 2: AST corresponding to the code of Listing 1

process can be seen as a new form of obfuscation). 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 isomorphic subgraphs between the malicious and benign ASTs. Since malicious obfuscation is responsible for their differences, we first deobfuscated the malicious files, to get the original syntax which resembles more a benign AST than the obfuscated version. JSDetox [71] and box-js [12] are combined with a manual analysis for the deobfuscation process.

3 METHODOLOGY

HideNoSeek aims at automatically rewriting the AST of a malicious JavaScript input into an existing benign one while retaining the malicious semantics after execution. This section first provides a high-level overview of our system, before discussing its three main components, namely an abstract code representation part, a clone detector, and a malicious code generator, into more details.

3.1 Conceptual Overview

As illustrated by Figure 1, HideNoSeek takes a malicious seed m and a benign document b as input, and outputs a sample s with the same AST as b, while retaining the malicious semantics of m.

First, we perform a static analysis of JavaScript documents, augmenting the traditional AST with control and data flow information, which we store in a joint structure, namely a PDG [21] (stage 1 of Figure 1). This structure enables to reason about the order in which statements are executed, as well as the conditions that have to be met for a specific execution path to be taken (Section 3.2). HideNoSeek then uses the previous graph structure to look for identical sub-ASTs between the malicious seed and the considered benign input (stage 2 of Figure 1). For this purpose, HideNoSeek looks for pairs of matching benign and malicious nodes (i.e., same abstract syntactic structure) and slices backward along their control and data flow as long as it reports further matching statements. These common structures are stored together in a list (slice), which we refer to as clones. Since a malicious sub-AST may be found several times in a given benign input, we define criteria to, e.g., maximize the clones' size or minimize the distance between the nodes inside a clone, thereby reducing the adjustment surface (Section 3.3). As a matter of fact, HideNoSeek replaces the benign clones by the malicious ones, and follows the original benign data dependencies, so as to automatically adjust the initial benign nodes–which were impacted by the replacement process–for them to still respect the initial benign AST structure, while keeping the malicious semantics after execution (stage 3 of Figure 1). Finally, we transform the AST back to code (Section 3.4).

This way, HideNoSeek crafted a sample, which has the AST of an existing benign input, while retaining the semantics of a malicious file, thereby foiling the detectors from Section 2.2.

3.2 Program Dependency Graph Analysis

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

3.2.1 Syntactic Analysis. The syntactic analysis is performed by the JavaScript parser Esprima [28], which takes a valid JavaScript sample as input and produces an ordered tree (AST) describing the syntactic structure of the program. Overall Esprima can produce 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* (except for *ContinueStatement* and *BreakStatement*). Figure 2 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.

3.2.2 Control Flow Analysis. Contrary to the AST, the CFG allows to reason about 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 flow 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

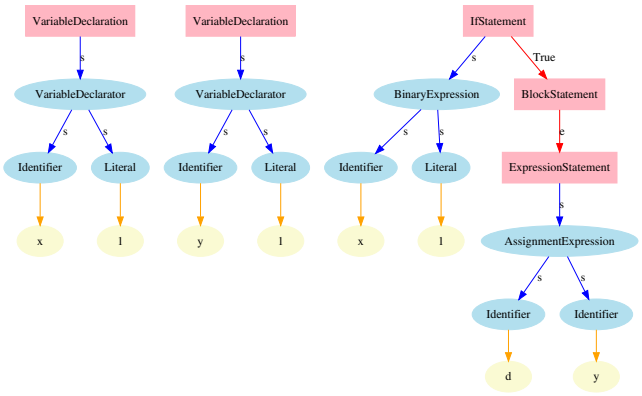

Figure 3: AST of Listing 1 extended with control flow

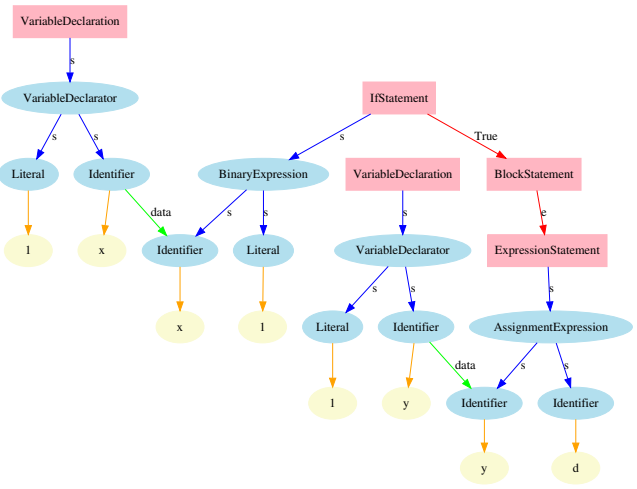

Figure 4: AST of Listing 1 extended with control & data flow

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

Listing 1: JavaScript code example

statements, but also contains non-statement and still non-terminal information, as shown in Figure 2, we first define a *statement dependency* (labeled with *s*) to refer to the edge between a statement node and its non-statement children, or between two non-statement nodes. 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 3 (for clarity reasons, we do not graphically represent the CFG or PDG, but add control and data flow information on the AST). Contrary to the AST of Figure 2, 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.

3.2.3 Data Flow Analysis. To this end, we implement a PDG [21], which augments the previous CFG with data flow. This code representation enables to bypass the sequencing choices made by the programmer to capture the data and control dependencies between the different program components. 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 4. This PDG indicates, in particular, the order in which statements from Listing 1 should be executed, e.g., as suggested by the data flow, lines 1 and 2 are executed before line 3; we could nevertheless interchange lines 1 and 2 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 in a local scope. To build our PDG, we traverse our CFG depth-first pre-order and maintain two variables lists. The first one contains the global variables, and the last one the local variables currently declared in the *considered block statement*, taking into account the specific local scope of variables defined with *let* or *const* (in the block where they were defined). For objects, 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. Thus, we consider the data flow 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 flow 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. Next, we handle functions' 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, we connect the function call nodes to the corresponding function definition nodes with a data dependency, thus defining the PDG at the program level [82].

3.3 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 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.

First, we aim at detecting sub-ASTs from a malicious file that can also be found in a benign one. We refer to such common structures as clones. To detect clones, we consider the algorithm of Komondoor et al. [45], which combines PDGs and a variation of program slicing [78]. First, we create equivalence classes (Section 3.3.1), which regroup common benign and malicious PDG statement nodes, based

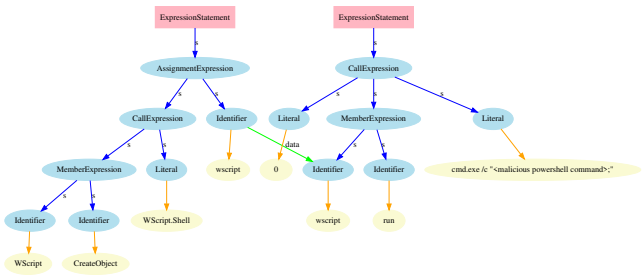

Figure 5: AST of Listing 2 extended with control & data flow

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

Listing 2: Malicious JavaScript code example

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

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

on their abstract syntactic meaning. Then, for each benign and malicious pair from the same class with the same statement dependencies (i.e., slicing criterion), we add them to the current clones list and slice backward along their control and data dependencies. We add these predecessors to the current clones list if and only if they match (same syntax), and iterate as long as matching statement nodes are found (Section 3.3.2). Finally, as a malicious sub-AST may be found several times in the same benign document, we define criteria to select the strongest clones (Section 3.3.3).

3.3.1 Equivalence Classes. Finding clones between a benign and a malicious file consists of finding isomorphic subgraphs between their abstract syntactic representations. Computing all pairs of benign/malicious statement nodes and comparing their syntax, to keep only the matching ones, would not be efficient. Thus, HideNoSeek first partitions the benign PDG statement nodes into equivalence classes based on their syntactic structure. For example, the PDG of Figure 4 would have four distinct classes: *VariableDeclaration* (with two elements), *IfStatement*, *BlockStatement*, and *ExpressionStatement*. Then, the equivalence classes are completed with the considered malicious file, e.g., the PDG of Figure 5 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 this example. We perform this test in the next step.

3.3.2 Clone Detection. The next step consists of iterating through the previous equivalence classes 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 control and data flow, starting from *b* and *m*, adding them as well as their predecessor to the current clones list 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. Because of backward slicing along the control and data flow, 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. 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 dependencies to find more non-contiguous clones. Because of this step, we can find 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, we keep a list 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 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;
    $benign\_parents \leftarrow$ backward_slice(benign);
    $malicious\_parents \leftarrow$ backward_slice(malicious);
    iterate over benign_parents and malicious_parents;
    call find_clone on the resulting combinations;
  end
else
  $benign\_parents \leftarrow$ backward_slice(benign);
  iterate over benign_parents;
  call once find_clone(benign_parent, malicious);
end
return clones_list

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

3.3.3 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. Still, for the attack to be effective, the *complete* malicious sample has to be replaced in syntactically equivalent parts of the considered benign file. The second criterion consists of maximizing the proportion of identical tokens between the benign and the crafted samples. Indeed, reproducing 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 3.4.2).

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 (up to three) different syntactic versions of a same malicious seed to improve the proportion of clones reported (c.f. Section 4.1.1). 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.

3.4 Malicious Code with a Benign AST

Once HideNoSeek has found identical benign and malicious sub-ASTs, it replaces the benign sub-ASTs with the corresponding malicious ones. This process then yields some adjustments for the crafted code to still be able to run. Therefore, HideNoSeek follows the data flow originating from the initial benign nodes (which have been replaced by their malicious equivalent) and modifies them, with respect to the AST, so that the malicious semantics is kept.

3.4.1 Clone Replacement. An AST is composed of inner and leaf nodes, the latter which represent the operands. Saying that a benign AST is identical to a malicious one means that they have the same nodes, with the same oriented edges. Still, the benign code is different from the malicious one, as the variables name are not directly contained in the AST, but are attributes of the leaves. Therefore, replacing the attributes of the benign leaves 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 3.3.2). Nevertheless, the replacement process has modified the benign environment and might interfere with the benign functionalities, which could result in the crafted sample not running anymore.

3.4.2 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 flow 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 flow points to, i.e., the object that HideNoSeek modified. 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 methods, such as *hasOwnProperty*, *toString*, which can be combined and are valid in all contexts.

Finally, the ECMAScript code generator Escodegen [70] is used to transform the modified AST back to JavaScript code.

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

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

4 COMPREHENSIVE EVALUATION

In this section, we outline the results of our extensive evaluation. To produce malicious samples with an existing benign AST, HideNoSeek uses 23 unique malicious seeds, whose ASTs it can rewrite in the ASTs of our 8,546 different benign scripts. We first evaluate the number of malicious samples our system was able to produce per seed, before considering the impact our attack would have. Then, we verify the validity and maliciousness of the samples previously crafted. Finally, we test HideNoSeek on real-world detectors and analyze its run-time performance.

4.1 Experimental Setup

The experimental evaluation of our approach rests upon two extensive datasets. The former contains 122,345 SHA1-unique malicious JavaScript samples and the latter 8,941 unique benign files.

4.1.1 Malicious Dataset. Our malicious dataset, presented in Table 1, 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) [37] and GeeksOnSecurity (GoS) [23], as well as the malware collection of Hynek Petrak (Hynek) [62] and the German Federal Office for Information Security (BSI) [10]. 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. Initially, JSDetox [71] and box-js [12] performed the samples' deobfuscation, but we could not automate the process 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, 18, 49, 64]. 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;[1] 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 executed. Therefore, they gave us four deobfuscated samples instead of two. 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 3.3.3, we kept the different variants (up to three for a given file) 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* seed. Besides, we are working at the AST level; therefore, we consider that two samples with the same AST but a different behavior are identical. After duplicate deletion, we retained 22 malicious seeds, to which we added a crypto-miner, as cryptojacking in browsers has become a widespread threat [30, 46, 77]. These 23 unique seeds represent in total 37 different syntactic variants. Finally, to verify to what extent our dataset was representative of the malicious distribution found in the wild, we extracted 13,884 samples from VirusTotal [73]. These were collected *after* the files we analyzed previously and did not contain duplicates. As before, we clustered them syntactically and got 8 clusters (Table 1), one file of each we deobfuscated. As the 8 deobfuscated samples matched our 23-sample set (7 matched exploit kits, and 1 a dropper), we deemed our dataset to be saturated.

[1]We discuss possible drawbacks induced by the deobfuscation process in Section 4.4.1

Table 1: JavaScript malicious dataset description

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
BSI	2017-18	85,059	10	10
Sum	2014-18	122,345	61	55
VirusTotal	2017-18	13,884	8	8

Table 2: JavaScript benign dataset description

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

4.1.2 Benign Dataset. As for the benign dataset (Table 2), we statically scraped the start pages of Alexa top 10,000 websites, also including external scripts. Given the fact that we extracted this JavaScript 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 [76].

4.2 Evasive Samples Generation

HideNoSeek leverages the previous 23 malicious seeds to produce malware with an existing benign AST. In this section, we first report on the samples our system could craft, before evaluating our attack's impact on the highest ranked web pages and libraries.

4.2.1 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 4.1.1), 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 3. This table represents the number of malicious samples crafted per malicious seed (#Samples), the number of nodes that HideNoSeek had to adjust due to the replacement of benign sub-ASTs with syntactically equivalent malicious ones (#Adjust), as well as the average number of nodes contained in the crafted samples (#Nodes)–i.e., the average number of nodes in the benign samples whose ASTs were reproduced by a given malicious seed. In particular, we make a distinction between the samples crafted with the benign AST of a top 1k web page, against a top 10k 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 10k web pages, but in practice we got 10% less; for Crypto-miner we even got 65% less than expected). 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 10k. It is rather the opposite since, on average, our PDGs contain more nodes for pages from Alexa top 10k than Alexa top 1k. Nevertheless, the first 1,000 seem to have a more complicated structure with, in particular, more data flow: 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[2] 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 is identical to benign scripts'. With the exploit kit Misc, HideNoSeek was able to generate an evasive sample for 78% of the pages from Alexa top 10k. On the contrary, it had difficulty to craft samples for the malicious seed Dropper and was unable to produce any outputs for RIG2. For both of them, the difficulty lay in the syntactic structures they used, which were practically never present in benign documents. For example, our dropper initially 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 three times in the same benign data. Thus, we looked for a new syntactic construct, semantically equivalent, 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. We found the following structure *a.b("")* in 118,700 statements that matched benign and malicious documents. As a consequence, we replaced the previous malicious dropper's construct with its equivalent *WScript.CreateObject("object")* and could this time hide the malicious seed in ten benign documents. Nevertheless, our tool crafted 4,735 samples for the PowerShell seed (Table 3), which is a dropper too. Therefore, an attacker could still hide a dropper in 4,735 web pages from Alexa top 10k. As for RIG2, it contained complex syntactic structures, such as *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 improvements for this process in Section 5.

Overall, and out of the 23 malicious seeds HideNoSeek got as input, it was able to craft malware for 22 of them. In total, it produced 9,725 malicious samples with benign ASTs of Alexa top 1k web pages and 91,020 for the top 10k. We believe this number can be improved by using different syntactic structures for the seeds: as shown above, we can identify the most common structures between the benign and malicious datasets, and therefore envision that the malware authors could adjust their code to use those constructs.

4.2.2 Impact of the Attack. HideNoSeek can hide a given malicious seed into different web pages–while keeping their initial AST–, thereby static detectors would fail to see the maliciousness. Next, 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 4 indicates how many malicious seeds HideNoSeek could hide in Alexa top 10. In particular, we could hide 78% of our seeds in the start pages of the two most visited web pages, which would maximize the impact of our attack, in terms of infected users. Except for *wikipedia.org*, where HideNoSeek did not report any clone[3], we could hide on average more than 61% 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 [43], 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 [76], and studied the number of malware we could hide inside (Table 5). The proportion ranges from 22% to 74% and is a bit lower

Table 3: Number of samples crafted per malicious seed, number of nodes that had to be adjusted (#Adjust) and average number of nodes contained in the crafted samples (#Nodes)

	ALEXA-1k			ALEXA-10k		
Seed	#Samples	#Adjust	#Nodes	#Samples	#Adjust	#Nodes
Blackhole1	558	101	106,897	5,040	76	120,189
Blackhole2	558	157	106,897	5,041	100	120,224
Crimepack1	568	45	107,991	5,424	38	117,841
Crimepack2	532	30	113,575	5,072	35	123,490
Crimepack3	127	74	140,399	1,364	97	156,408
Crypto-miner	90	203	80,943	311	119	133,927
Donxref1	629	90	102,710	5,888	36	112,674
Donxref2	454	262	117,727	4,233	221	133,895
Dropper	0	0	0	10	142	153,840
EK	683	27	96,434	6,487	6	104,422
Fallout	427	143	120,639	3,732	67	137,716
Injected1	680	72	97,069	6,447	54	105,217
Injected2	502	131	117,838	4,810	66	128,429
Meadgive	535	55	112,639	5,106	47	122,856
Misc	683	27	96,434	6,487	6	104,431
Neclu1	300	93	122,893	2,890	110	144,388
Neclu2	530	59	113,118	5,031	64	123,925
Packer	204	45	126,836	2,325	48	138,996
PowerShell	507	24	110,891	4,735	22	123,046
RIG1	23	124	179,150	244	171	170,135
RIG2	0	-	-	0	-	-
VBScript1	584	15	100,254	5,275	8	114,502
VBScript2	551	49	105,198	5,068	16	118,696

Table 4: Number of samples that HideNoSeek hid in Alexa top 10 and number of nodes the crafted samples contain

Rank	Domain	#Samples	#Nodes
1	google.com	18	58,322
2	youtube.com	18	151,527
3	facebook.com	13	143,772
4	baidu.com	8	35,018
5	wikipedia.org	0	90
6	qq.com	14	54,450
7	yahoo.com	14	67,264
8	taobao.com	19	89,910
9	tmall.com	17	63,102
10	amazon.com	17	36,060

[2]We define the process of *hiding* a malicious seed in a benign sample to refer to the rewriting of the malicious AST into an existing benign one

[3]The start page contained almost no JavaScript code; therefore its PDG only had 90 nodes, which is, e.g., 648 times less than *google.com*, and prevented HideNoSeek from finding any matches with malicious seeds

Table 5: Analysis of the number of seeds successfully hidden among the most widely used JavaScript libraries [76]

Library	Alexa usage	Most common version	#Seeds	#Nodes
jQuery	73.5%	1.12.4	17	35,511
Bootstrap	18.1%	3.3.7	12	10,973
Modernizr	11.4%	2.8.3	5	3,174
MooTools	2.4%	1.6.0	15	27,786
Angular	0.4%	1.7.5-min	17	60,234

than from Alexa top 10, where on average 14 seeds could be hidden, compared to 13 in the libraries. Similar to Android malware in repackaged applications [9, 65, 85], 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 [76].

4.3 Validity Tests

Based on the insights that HideNoSeek could leverage 22 out of 23 malicious seeds to craft 91,020 malicious scripts, which have the same AST as scripts extracted from Alexa top 10k, in this section, we verify the validity and maliciousness of the produced samples.

4.3.1 Same AST for Crafted and Benign Scripts. First and by construction, all HideNoSeek crafted samples have the same AST as the benign scripts it used for the replacement process. Without further testing, this guarantees that classifiers purely based on syntactic features (e.g., JaSt [18]) are not able to distinguish them.

4.3.2 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 (Section 3.3.3). On average, 0.29 token differ for each 91,020 file crafted from Alexa top 10k websites (containing on average 127,693 nodes). Depending on the detector our implementation is targeting, this may be sufficient to prevent the evasion; e.g., for Kizzle [69], our malicious sample would be clustered together with benign samples due to the choice of maximum distance within a cluster. Also, we would produce *at most* three such samples every ten crafted script, which we assume to be negligible when considering the impact of our attack (e.g., 73.5% of the websites use jQuery and a malicious modification of the most widely used version 1.12.4 would affect 29.7% of these sites [76]).

4.3.3 Crafted Scripts Still Running. Third, HideNoSeek rewrote the ASTs of malicious JavaScript inputs into existing benign ones, which could result in the crafted samples not running. To decrease the proportion of broken samples, we implemented a module, which detects the program's parts impacted by our transformations–by following the data flow originating from our replacements (Section 3.4.2). Still, some adjustments may not be working in the 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 for clones between a benign and a malicious input. Nevertheless, it still is valid to replace two independent benign sub-ASTs by two syntactically equivalent malicious ones, with variables declared in the global scope, depending on one another. In this case, we have to ensure that the execution order of the variables is respected to avoid a *ReferenceError* at run-time. To verify the correct execution of our crafted samples, we used the library *jsdom* [35]–which emulates web browser functionalities, e.g., DOM elements–to test JavaScript implementations using web standards with *Node.js*. This way, we could ensure that we did not break functionality that requires DOM components. However, this environment cannot be used to test scripts scraped from websites, as the JSDOM is a placeholder, and the downloaded JavaScript often relies 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 1,631 samples we crafted from jQuery, 1,175 were still able to run (72%). In addition, we could hide 19/23 malicious seeds in jQuery libraries.[4] With the exception of Crimepack3 (accounting for two jQuery crafted files), each seed had at least one running sample. As stated in Section 4.3.2, 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, 51, 75, 80] 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 1,175 malicious working versions of jQuery, which had the same AST as the original ones.

4.3.4 Crafted Scripts with a Malicious Behavior. Finally, besides verifying the executability of the crafted samples, we checked their maliciousness. To this end, we randomly selected 5 working crafted samples per malicious seed[5] and analyzed them in two dimensions.

First, we executed these 88 crafted scripts in a web browser and manually verified which malicious statements were already called during initialization, and which required manual calling. In particular, the jQuery library defines objects and methods for future use and does not necessarily directly call them. As a consequence, we searched for all malicious parts in the considered scripts and instantiated, e.g., the functions or objects–with correct parameter values–required to execute the malicious functionalities. Still, if the malicious part was defined, e.g., within a closure, we could not call it. In addition, if the considered malicious element was in an *if condition*, for the sake of simplicity, we first changed the condition to *true*. Out of the 88 samples we analyzed, we validated the correct execution and the expected malicious behavior of 59 (67%).

Second, changing the *if conditions* to *true* is not acceptable as it changes the AST of the crafted samples, while our attack aims at reproducing existing benign ASTs. Still, given our attacker model, attackers can freely modify the code as long as the AST does not change. Hence, they can either directly manipulate the conditions that have to evaluate to *true* for their attack to be effective, or change the environment prior to the conditions (e.g., the value of a variable defined before the condition and used in it). For this purpose,

[4] In Section 4.2.2, we could hide 17 seeds in jQuery version 1.12.4 specifically, while we consider here all different jQuery versions

[5] We considered the 18 seeds from Section 4.3.3 (which generated running samples), one of which only had 3 working samples, therefore we tested 88 crafted samples

HideNoSeek includes a module which slices backward along the control flow of a modified node and outputs the different conditional statements that need to evaluate to *true* for the malicious code to be executed. In particular, we applied this module on the previous 59 scripts and analyzed their conditional statements. To change them to true, without modifying the AST, we followed the five following rules. First, the *and* and *or* operators are both `LogicalExpressions` and can therefore be interchanged. Second, the comparison operators `==`, `!==` and `===` are all `BinaryExpressions` and can also be interchanged. Still, this is not the case for the operator `!`, which is an `UnaryExpression`. Thus, we need both an element, which directly evaluates to true and another one to false. Hence the third point: any object whose value is not *undefined* or *null* evaluates to true [56], e.g., *window*, *String*. These objects can then be combined with the functions, properties, and methods used in Section 3.4.2. Fourth, *undefined* evaluates to false, therefore also calling a non-existing property on an object, e.g., *console.foo*. We cannot leverage the *undefined* property to call a method on an object though. In this case, we can use *window.Boolean(false)*. Last but not least, to access a property of an object, both bracket and dot notations can be used [55] and are syntactically equivalent. Finally, we combined these rules on the previous 59 scripts and could modify all the required conditions so that they always evaluate to true.

4.4 Evaluation Against Real-World Systems

In this section, we evaluate the camouflaging of malicious samples by HideNoSeek in practice. We first target traditional structural-based detectors before focusing on lexical and syntactic classifiers.

4.4.1 Evading Structural-Based Detectors. To exactly reproduce a benign AST, we first deobfuscated the malicious seeds, thus leaving their malicious logic in the open. Therefore, signatures might be able to detect our crafted samples. For this purpose, we manually studied the Yara rules [84] built for malicious JavaScript detection. These rules are based on strings and aim at describing patterns-based malware families. In particular, the rules targeting exploits, obfuscation, and exploit kits would not work on HideNoSeek crafted samples thanks to our deobfuscation process. As a matter of fact, the regular expressions considered try to match calls to *eval*, *unescape*, special encodings or specific identifiers (which we renamed during the deobfuscation process), that are not present in the deobfuscated version anymore. Nevertheless, some rules could still detect JavaScript implementing CVEs, e.g., whenever the reference to a specific instantiated Java object is written in plain text. Yet, in this case, string splitting would foil the signature. In practice, VirusTotal analyzed the 88 samples from Section 4.3.4 with between 48 and 60 AV-systems. 8 crafted samples were detected (*Donxref1* 5 times and *PowerShell* 3 times), 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 bypass signature-detection. After such modifications, we could still produce 211 samples for *Donxref1* and 795 for *PowerShell*, but this time they evaded VirusTotal detection.

Initially, attackers focused on obfuscation to significantly complicate the detection and analysis of malware. Still, by doing so, they added specific and recurrent malicious patterns to their JavaScript files. As a consequence, analyses directly based on the code were able to leverage these differences for an accurate detection. Deobfuscation now enables to evade most detections by such systems (Section 4.4.2). In addition, HideNoSeek can be seen as a new form of obfuscation, which this time does not add specific patterns to the files it modifies. Even though it may adjust some nodes with generic transformations (Section 3.4.2), it does not change the overall code syntax. Implementing a system, which would know HideNoSeek's internals and recognize, e.g., the function's names we changed, such as *toString*, or *isFinite*, or integrate a new set of rules targeting deobfuscated malicious JavaScript, e.g., *ActiveXObject*, is deemed to produce a lot of false positives (c.f. Section 4.4.2).

4.4.2 Evading Lexical and Syntactic Classifiers. Besides evading structural-based detectors, HideNoSeek is also effective against lexical and syntactic classifiers. In this section, we train a model on benign and malicious samples found in the wild before evaluating its accuracy on HideNoSeek crafted samples, also assessing the classification of the benign samples used for the camouflage. As a second scenario, we update our previous model with HideNoSeek samples and perform the same experiments as before.

Classifiers Training. First, we built machine learning models–each one of them containing 10,000 unique malicious files and as many benign ones–to train the classifiers selected to test HideNoSeek samples. In the first scenario (S1), we randomly picked 8.17% of the malicious samples from each of our malware providers (DNC, Hynek, GoS and BSI, Section 4.1.1), to have a malicious dataset containing 10,000 files, representative of the distribution found in the wild, through our multiple sources and the respect of their initial proportions (Table 1). For the benign part, we randomly selected 5,000 unique benign samples from our Alexa dataset and 5,000 from Microsoft products,[6] to have both web and non-web JavaScript. We deemed our model to be balanced, with as many malicious as benign samples. To improve the experiments' reproducibility and limit effects from randomized datasets, we repeated the previous procedure 5 times and averaged the detection results.

In the second scenario (S2), we leveraged the fact that machine learning-based detectors are able to *learn*. In particular, since the detectors we consider are based on machine learning algorithms, our evaluation should also use machine learning models, trained with samples crafted by HideNoSeek. For this purpose, we built 5 new models. We randomly selected 5,000 malware from our previous models and added 5,000 randomly selected samples from our crafted scripts, to illustrate the adaptation of the models to inputs that may be found someday in the wild (i.e., documents produced by HideNoSeek). We kept the same benign sets as previously.

Evasion in Practice. We now consider exclusively the model S1 for classification purpose. HideNoSeek is a novel camouflage attack that rests upon the assumption that malicious obfuscation leaves traces in the syntax of malware, which lexical and syntactic classifiers leverage for an accurate detection. Hence, perfectly mapping a benign AST (while retaining the malicious semantics) will automatically foil most of these detectors. At the same time, some systems also consider identifiers value or type information of AST nodes, e.g., for CSP generation [58], vulnerability detection [68], as well as in Zozzle for malicious JavaScript detection [16].

[6] Microsoft Exchange 2016 and Microsoft Team Foundation Server 2017

To verify the evasion capability of HideNoSeek in practice, we used the open source tool JaSt [2], a reimplementation of Cujo [64] and a reimplementation of Zozzle[7] [16] (c.f., Section 2.2) to classify our crafted samples. We first focussed on classifying one malicious sample extracted from each of the 61 clusters obtained in Section 4.1.1. JaSt accurately detected 52 samples on average, Cujo 51.4 and Zozzle 44. We manually looked at the false negatives, which were either not obfuscated or just using a call to *unescape* (translated by a function call with a string parameter) and explains why these tools did not perform as well as in their corresponding papers. Also, and more specifically for Zozzle which includes identifiers' name information, the detectors may struggle to classify inputs whose syntactic structures they had never seen before; this may happen as the 61 clusters have a different syntax and may not all be represented in the training set. As a next step, we classified the 37 different deobfuscated variants (Section 4.1.1) obtained from the previous samples. This time, JaSt recognized only 1 of them as malicious, Cujo 0.4 and Zozzle 2. These results are in line with the assumption that malicious obfuscation induces specific patterns in the files, while deobfuscated samples have a more benign-looking structure, which can thereby also be found in benign documents.

Finally, we classified the 118,052 samples[8] produced by HideNoSeek. By construction, our attack foils JaSt, which purely relies on the AST for malicious JavaScript detection. Still, it could detect 26.6 crafted samples (accuracy: 0.02%). Since these files have the same AST as the original benign samples, the 6.2 benign documents (on average) used to produce the previous 26.6 samples are false positives. On the contrary, Zozzle also includes the text of the AST node as an additional feature. Still, it could detect 3.8 crafted samples (accuracy: 3.2E-3%), which used 0.8 benign files for the hiding process. As none of them were classified as malicious (false positives), 0.8 scripts changed classification between the benign and the malicious variants (6.8E-4% of the crafted samples). Thereby, HideNoSeek can also evade syntactic detectors using nodes' value as additional features. Similarly, Cujo detected 15.8 crafted samples (accuracy: 0.01%), leveraging 7.6 benign files for the camouflaging process, 6.6 of which are false positives. Thus, 1 script changed classification between the benign and the malicious versions (8.5E-4% of the crafted samples). Thereby, we assume that the tokens which might differ between the two file versions have a negligible impact on the overall files' classification.

Evasion of Retrained Classifiers. As a second scenario, we updated our previous models with HideNoSeek samples (model S2) to illustrate the fact that machine learning-based detectors are able to *learn*. Contrary to the previous experiments, JaSt was this time able to detect 108,164.4 crafted samples[9] (accuracy: 95.68%). Nevertheless, the 5,907.8 benign files used to hide the previous malware were all misclassified as malicious. In fact, we trained the detectors with malicious samples, which have the same AST as benign samples. Therefore, they could not learn any features specific to one class and had to decide whether to classify all such files as malicious (thereby Alexa samples would be false positives, actual case) or as benign (thereby HideNoSeek crafted samples would be false negatives). We observe a similar trend for Zozzle and Cujo, which accurately detected 103,513.4 and 108,489 crafted samples respectively (accuracy: 91.56% and 95.96%). The true positives from Zozzle leveraged 5,695 benign inputs for the hiding process, 5,054 of which it classified as false positives (88.74%). Hence 641 samples changed classification between the benign and malicious variants (0.57% of the crafted samples). Similarly, Cujo reported 377 classification changes (0.3% of the crafted samples). For both tools, even if the proportion of classification changes is higher than with S1, it is still too low to provide a useful and reliable defense mechanism against HideNoSeek. Therefore, considering the nodes' value of the AST or the tokens value (including type information) does not enable to accurately detect HideNoSeek samples.

[7]With automatic features selection, 1-level features and multinomial Naive Bayes, which provided a detection rate similar to the original paper's, see Appendix A

[8]We consider here all crafted samples, i.e., also produced by variants from the same seed

[9]We excluded the samples used to train the detectors from their corresponding test set; therefore we classified 113,052 samples in this section

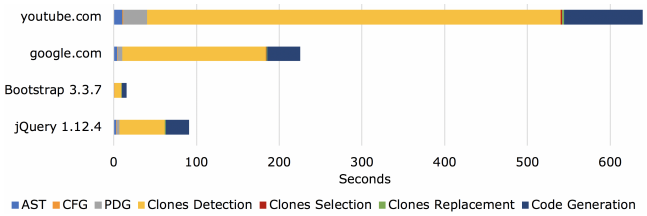

Figure 6: Time required to test our 23 malicious seeds on the two most popular websites and libraries

4.5 Run-Time Performance

We tested HideNoSeek's run-time performance 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 6 presents the processing times, for all stages of HideNoSeek, to craft the 65 previous malware (Section 4.2.2). The most time-consuming operation is the clones detection, which is NP-complete (Section 3.3) and highly depends on the PDGs' size (Table 4, Table 5). The code generation phase is also 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 65 samples took sixteen minutes.

5 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.

5.1 Limitations

HideNoSeek is based on a static analysis of JavaScript to build both the control and data flow in a given script. On the one hand, this approach provides 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, 20, 33, 34]. 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. In addition, JavaScript uses prototype chaining [54] to model inheritance, where properties can be added or removed during the execution, and property names may be dynamically computed. 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 malicious payload at their disposable).

5.2 Existing Defenses

As mentioned in Section 1, 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 black-box access to the classifier or dataset related inputs [8, 13, 14, 19, 75]. 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 [67], statistical tests [25], dimensionality reduction [7, 79], the detection of adversarial perturbations [52, 53], or vectorization [36]. Nevertheless, we envision that none of them would work for our attack as we perfectly reproduce an existing benign syntax, instead of merely injecting benign features.

5.3 Potential Detection Strategies

A possibility to detect HideNoSeek samples would be to (a) recognize the original benign input used for the hiding process, and (b) notice that it differs from the benign file it is supposed to be. If the original sample is recognized, a checksum test should indicate whether it is the original version or not. Still, the official library source code can be altered for benign purposes, like functionality extensions or caching proxies, or stored together with other libraries. In particular, we used *retirejs* [57] to extract 73 different versions of jQuery used by Alexa top 10k. Still, none of them matched the hash given on the official jQuery web page, because they were either combined with other JavaScript code or contained, e.g., the name of the caching proxy, essentially nullifying a checksum. In practice, hash testing is, therefore, not a reliable solution against our attack.

Apart from this and since HideNoSeek adjusts the code it crafted with calls to, e.g., *toString* or *isFinite*, it can create seemingly dead code (due to the lack of proper usage of the result) whose frequency could be an indicator of our crafted scripts. Still, relying on such artifacts is likely to produce a lot of false positives (Section 4.4), as also evidenced by, e.g., Google sites making extensive use of parameterless *toString* invocations [63]. Similarly, given the quality of JavaScript code in the wild, our experience leads us to believe that otherwise useless variables are not a good indicator of HideNoSeek crafted samples either: for example when calls to *console.log* are commented out without removing the assignments of variables only used in that call.

Last but not least, HideNoSeek is an attack against *static* malicious JavaScript detectors, and does not necessarily foil hybrid or dynamic detectors such as Rozzle [44] or J-Force [42], which force the JavaScript execution engine to test all execution paths systematically. Similarly, Revolver [40] could detect that the original benign and the crafted samples have the same AST but its dynamic detector would classify the samples differently, hence detecting the evasion attempt.[10] Still, dynamic detectors are usually slow, thus rather work on a pre-filtered list of samples likely to be malicious [11]. Since this list tends to be generated by static systems–which are much faster–, e.g., lexical or syntactic, HideNoSeek crafted samples may not even be dynamically analyzed.

5.4 Improving the Evasion

To improve the number of malicious samples HideNoSeek can craft, 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.

6 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. Also, it does not try to statistically enhance the proportion of benign features in a malicious file, but exactly reproduces actual benign JavaScript ASTs, which by construction foils most static detectors. 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.

6.1 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. assessed the security of learning-based detection techniques by studying the range of possible attacks, according to the information leaks an attacker might have [75]. In addition, they explored the strategy of training a substitute model to find evasive inputs, as well as the possibility to modify a malicious file so that it mimics the features of a chosen benign target [74]. 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 [22]. 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 required) [17, 80]. Contrary to the previous approaches, Maiorca et al. aimed at injecting malicious content in benign PDF documents so as to introduce

[10] We contacted the authors to test our samples on Revolver, but the system is not available anymore, which prevented this experiment

minimum differences within its benign structure, while having a malicious behavior (reverse mimicry) [51]. Last but not least, Grosse et al. adapted the algorithm of Papernot et al. [61]–initially defined for images–to find which features should be changed to craft adversarial samples in the malware field [26].

6.2 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 [29]. 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 flow 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 [83]. 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 flow [82]. This new data structure was also used by Backes et al. to identify different types of Web application vulnerabilities [3].

6.3 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 [47]. 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 [48]. Last but not least, Komondoor et al. combined PDGs with the use of program slicing to find clones in C programs [45]. The addition of the slicing part enabled them to find non-contiguous clones, clones in which matching statements had been reordered, as well as clones intertwined with each other.

7 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 rewrites the ASTs of malicious samples into existing benign ones. The key elements of our approach are the following: (a) a modeling of the control and data flow extracted from the malicious seeds to rewrite, and from the benign files whose ASTs we perfectly reproduced; (b) a detection and analysis of isomorphic sub-ASTs, with respect to control and data dependencies, between the previous benign and malicious inputs and (c) the replacement and adjustment of benign sub-ASTs by their malicious equivalents.

We evaluated HideNoSeek on an extensive dataset of both malware and benign scripts. In practice, our approach is highly effective with its production of 91,020 malware from 22 malicious seeds and 8,279 benign web pages. In addition, it has a high impact: on average HideNoSeek produces 14 different malicious samples with the same AST as each Alexa top 10 and 13 for each of the five most popular JavaScript libraries. As far as evasion is concerned, HideNoSeek is highly effective. When the targeted systems have no knowledge about our crafted samples, they misclassify them 99.98% of the time (false negatives). In the case that defenders are aware of HideNoSeek and trained their detectors with such crafted samples, around 91.56% are accurately detected, but in return over 88.74% of the benign inputs, whose ASTs we reproduced, are false positives. As a consequence, most of the static detectors relying on syntactic, lexical or structural features are impacted by our attack, which makes them misclassify input samples over 88.74% of the time, rendering them inapt to handle HideNoSeek samples.

ACKNOWLEDGMENTS

We would like to thank the German Federal Office for Information Security (BSI), VirusTotal and Kafeine DNC, which provided us with malicious JavaScript samples for our experiments. We would also like to thank the anonymous reviewers of this paper for their well-appreciated feedback. Special thanks also go to Nils Glörfeld and Dennis Salzmann, who respectively contributed to JavaScript deobfuscation and a manual analysis of our crafted samples. Furthermore, we gratefully acknowledge the help in terms of feedback or inspiring discussion of Christian Rossow, Konrad Rieck, Marius Steffens, and Pierre Laperdrix. 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).

REFERENCES

[1] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.
[2] Aurore54F. [n.d.]. JaSt - JS AST-Based Analysis. In: https://github.com/Aurore54F/JaSt. Accessed on 2018-12-17.
[3] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *EuroS&P*.
[4] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. 2010. The Security of Machine Learning. *Machine Learning* (2010).
[5] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. 2006. Can Machine Learning Be Secure?. In *ASIACCS*.
[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *International Conference on Software Maintenance (ICSM)*.
[7] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. 2018. Enhancing Robustness of Machine Learning Systems via Data Transformations. *Annual Conference on Information Sciences and Systems (CISS)* (2018).
[8] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2009. Multiple Classifier Systems for Adversarial Classification Tasks. In *International Workshop on Multiple Classifier Systems*.
[9] Joany Boutet. [n.d.]. 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] BSI. [n.d.]. German Federal Office for Information Security (BSI). In: https://www.bsi.bund.de/EN. Accessed on 2019-07-18.

[11] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *International Conference on World Wide Web (WWW)*.

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

[13] Nicholas Carlini and David Wagner. 2017. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In *ACM Workshop on Artificial Intelligence and Security*.

[14] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *S&P*.

[15] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *International Conference on World Wide Web (WWW)*.

[16] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security*.

[17] Hung Dang, Yue Huang, and Ee-Chien Chang. 2017. Evading Classifiers by Morphing in the Dark. In *CCS*.

[18] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *DIMVA*.

[19] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2015. Analysis of Classifiers' Robustness to Adversarial Perturbations. *Machine Learning* (2015).

[20] Asger Feldthaus and Anders Møller. 2013. Semi-Automatic Rename Refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

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

[22] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. 2006. Polymorphic Blending Attacks. In *USENIX Security*.

[23] GeeksOnSecurity. [n.d.]. Malicious Javascript Dataset. In: https://github.com/geeksonsecurity/js-malicious-dataset. Accessed on 2018-07-13.

[24] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

[25] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. 2017. On the (Statistical) Detection of Adversarial Examples. *arXiv preprint arXiv:1702.06280v2* (2017).

[26] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. In *European Symposium on Research in Computer Security*.

[27] Yongle Hao, Hongliang Liang, Daijie Zhang, Qian Zhao, and Baojiang Cui. 2014. JavaScript Malicious Codes Analysis Based on Naive Bayes Classification. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*.

[28] Ariya Hidayat. [n.d.]. ECMAScript Parsing Infrastructure for Multipurpose Analysis. In: http://esprima.org. Accessed on 2018-09-16.

[29] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security*.

[30] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. 2018. How You Get Shot in the Back: A Systematical Study About Cryptojacking in the Real World. In *CCS*.

[31] Fraser Howard. [n.d.]. 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.

[32] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. 2012. EvilSeed: A Guided Approach to Finding Malicious Web Pages. In *S&P*.

[33] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis*.

[34] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *International Symposium on Static Analysis (SAS)*.

[35] jsdom. [n.d.]. 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.

[36] Vishaal Munusamy Kabilan, Brandon Morris, and Anh Nguyen. 2018. VectorDefense: Vectorization as a Defense to Adversarial Examples. *arXiv preprint arXiv:1804.08529v1* (2018).

[37] Kafeine. [n.d.]. MDNC - Malware don't need coffee. In: https://malware.dontneedcoffee.com. Accessed on 2018-09-27.

[38] Alex Kantchelian, J. Doug Tygar, and Anthony D. Joseph. 2016. Evasion and Hardening of Tree Ensemble Classifiers. In *International Conference on Machine Learning*.

[39] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. 2011. "NoFus: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code". In *Microsoft Research Technical Report*.

[40] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, and Christopher Krügel and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *USENIX Security*.

[41] Zeinab Khorshidpour, Sattar Hashemi, and Ali Hamzeh. 2017. Evaluation of Random Forest Classifier in Security Domain. *Applied Intelligence* (2017).

[42] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *WWW*.

[43] Yonathan Klijnsma. [n.d.]. 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.

[44] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *S&P*.

[45] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *International Symposium on Static Analysis (SAS)*.

[46] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *CSS*.

[47] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *Working Conference on Reverse Engineering*.

[48] Jens Krinke. 2001. Identifying Similar Code with Program Dependence Graphs. In *Working Conference on Reverse Engineering (WCRE)*.

[49] Pavel Laskov and Nedim Šrndić. 2011. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *Annual Computer Security Applications Conference (ACSAC)*.

[50] Daniel Lowd and Christopher Meek. 2005. Adversarial Learning. In *International Conference on Knowledge Discovery in Data Mining (KDD)*.

[51] Davide Maiorca, Igino Corona, and Giorgio Giacinto. 2013. Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *ASIACCS*.

[52] Dongyu Meng and Hao Chen. 2017. MagNet: A Two-Pronged Defense Against Adversarial Examples. In *CCS*.

[53] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. 2017. On Detecting Adversarial Perturbations. In *International Conference on Learning Representation (ICLR)*.

[54] Mozilla Developer Network. [n.d.]. Inheritance and the prototype chain. In: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain. Accessed on 2019-01-31.

[55] Mozilla Developer Network. [n.d.]. Property accessors. In: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_accessors. Accessed on 2019-05-19.

[56] Mozilla Developer Network. [n.d.]. Standard built-in objects/ Boolean. In: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean. Accessed on 2019-05-18.

[57] Erlend Oftedal. [n.d.]. Retire.js: What you require you must also retire. In: https://retirejs.github.io/retire.js/. Accessed on 2018-10-31.

[58] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In *CCS*.

[59] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. 2016. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv preprint arXiv:1605.07277* (2016).

[60] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks Against Machine Learning. In *ASIACCS*.

[61] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Euro S&P*.

[62] Hynek Petrak. [n.d.]. Javascript Malware Collection. In: https://github.com/HynekPetrak/javascript-malware-collection. Accessed on 2018-07-17.

[63] PublicWWW. [n.d.]. `toString()` Usage. In: https://publicwww.com/websites/%22toString%28%29%22/. Accessed on 2019-07-18.

[64] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Annual Computer Security Applications Conference (ACSAC)*.

[65] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In *Annual Computer Security Applications Conference (ACSAC)*.

[66] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF Detection using Metadata and Structural Features. In *Annual Computer Security Applications Conference (ACSAC)*.

[67] Charles Smutz and Angelos Stavrou. 2016. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *NDSS*.

[68] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*.

[69] Ben Stock, Benjamin Livshits, and Benjamin Zorn. 2016. Kizzle: A Signature Compiler for Detecting Exploit Kits. In *Dependable Systems and Networks (DSN)*.
[70] Yusuke Suzuki. [n.d.]. ECMAScript Code Generator. In: https://github.com/estools/escodegen. Accessed on 2018-06-15.
[71] Sven. [n.d.]. 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.
[72] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2017. The Space of Transferable Adversarial Examples. *arXiv preprint arXiv:1704.03453v2* (2017).
[73] VirusTotal. [n.d.]. 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.
[74] Nedim Šrndić and Pavel Laskov. 2013. Detection of Malicious PDF Files Based on Hierarchical Document Structure. In *NDSS*.
[75] Nedim Šrndic and Pavel Laskov. 2014. Practical Evasion of a Learning-Based Classifier: A Case Study. In *S&P*.
[76] W3Techs. [n.d.]. Usage of JavaScript libraries for websites. In: https://w3techs.com/technologies/overview/javascript_library/all. Accessed on 2018-11-13.
[77] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. 2018. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *European Symposium on Research in Computer Security (ESORICS)*.
[78] Mark Weiser. 1981. Program Slicing. In *International Conference on Software Engineering (ICSE)*.
[79] Weilin Xu, David Evans, and Yanjun Qi. 2018. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *NDSS*.
[80] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *NDSS*.
[81] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. In *International Conference on Malicious and Unwanted Software (MALWARE)*.
[82] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *S&P*.
[83] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Annual Computer Security Applications Conference (ACSAC)*.
[84] Yara-Rules. [n.d.]. Repository of Yara rules. In: https://github.com/Yara-Rules/rules. Accessed on 2019-01-23.
[85] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *ACM Conference on Data and Application Security and Privacy*.

A REIMPLEMENTATION OF ZOZZLE

To evaluate the evasion capability of HideNoSeek on static detectors also including node values, we reimplemented Zozzle [16]. In particular, we extract features such that they contain the context in which they appear (AST node) and the corresponding node value. To select the features most predictive of malicious or benign intent, we use the χ^2 algorithm, such that $\chi^2 \geqslant 10.83$ (confidence of 99.9%, as Curtsinger et al. described in their paper).

First, we trained the classifier with the samples from scenario (S1), representative of real-world samples (Section 4.4.2). To verify if our results are similar to the paper's, we classified our benign and malicious JavaScript collections (excluding HideNoSeek samples at this stage). As we only retained 86% accuracy and over 78% false positives with Bernoulli Naive Bayes (BNB)–the classifier used in Zozzle–we tested Multinomial Naive Bayes (MNB), which also assumes all features to be statistically independent, and Random Forest (RF) that does not have this assumption. We assume that BNB did not perform as well as in the paper for two reasons. First, the authors focussed on heap spray payloads, while we have more diverse JavaScript samples, which may, therefore, have more different syntactic structures. Second, Zozzle is from 2011 and malicious JavaScript from that time may have been less obfuscated, in particular concerning identifiers renaming, than more recent samples.

Table 6: Comparison of Zozzle's accuracy on HideNoSeek samples depending on the classifier used

Tool	Zozzle-BNB	Zozzle-MNB	Zozzle-RF
on 61 clusters	61	44	43.6
on 37 seeds	37	2	2
(S1) - TP	1.26%	0.003%	0%
(S1) - #benign used in TP	409.4	0.8	0
(S1) - FP on benign from TP	97.41%	0%	0/0
(S1) - changed classification	9E-3%	6.8E-4%	0%
(S2) - TP	92.08%	91.56%	91.54%
(S2) - #benign used in TP	5,474.4	5,695	5,688.8
(S2) - FP on benign from TP	99.67%	88.74%	88.82%
(S2) - changed classification	0.016%	0.57%	0.56%

On the contrary, with MNB, we had less than 1% false positives and around 7% false negatives (w.r.t. the previously mentioned data set), which is similar to the paper's results. Also, MNB is still close to BNB: the former takes into account the frequency of a feature, while the latter considers its presence or absence. On the contrary, RF performed significantly better than the initial paper, with less than 1% false positives and false negatives. For this purpose, we performed the experiments from Section 4.4.2 with MNB (i.e., apply the learned models to HideNoSeek samples), but also present the results we would have had with BNB and RF in Table 6. We can see, in particular, that BNB classifies more samples as malicious than MNB and RF (for true positives as well as false positives), which is in line with our previous observation. MNB and RF have similar results: they recognize extremely few HideNoSeek crafted samples (less than 0.003%) in the scenario (S1), which was expected due to the benign-looking structures of our samples. On the contrary, they recognize most of our crafted samples as malicious in (S2)–which was trained on such samples–but at the same time misclassify the benign inputs we used to hide the detected malicious samples in (over 88%). Also, less than 0.57% changed classification between the initial benign samples and the malicious files reproducing their ASTs, which shows that HideNoSeek can bypass static detectors even when node values are adopted in the AST (a more thorough analysis is presented in Section 4.4.2).