JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code

Wei Xu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA
wxx104@cse.psu.edu

Fangfang Zhang
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA
fuz104@cse.psu.edu

Sencun Zhu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA
szhu@cse.psu.edu

ABSTRACT

The dynamic features of the JavaScript language not only promote various means for users to interact with websites through Web browsers, but also pose serious security threats to both users and websites. On top of this, obfuscation has become a popular technique among malicious JavaScript code that tries to hide its malicious purpose and to evade the detection of anti-virus software. To defend against obfuscated malicious JavaScript code, in this paper we propose a mostly static approach called *JStill*. JStill captures some essential characteristics of obfuscated malicious code by function invocation based analysis. It also leverages the combination of static analysis and lightweight runtime inspection so that it can not only detect, but also prevent the execution of the obfuscated malicious JavaScript code in browsers. Our evaluation based on real-world malicious JavaScript samples as well as Alexa top 50,000 websites demonstrates high detection accuracy (all in our experiment) and low false positives of JStill. Meanwhile, JStill only incurs negligible performance overhead, making it a practical solution to preventing obfuscated malicious JavaScript code.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security

Keywords

JavaScript, Obfuscation, Static Detection

1. INTRODUCTION

JavaScript based attacks have been reported as top Internet security threats in recent years [20] [14]. To defend against the malicious JavaScript code behind these attacks, most of the users rely on the protection provided by anti-virus software. Unfortunately, the effectiveness of static signature based anti-virus software is often thwarted by obfuscation techniques. In fact, malicious JavaScript code has been increasingly applying obfuscation techniques to evade the detection of anti-virus software and to hide its malicious intent. For example, many previously known ActiveX based attacks now adopt obfuscation to hide their exploits [24]. Besides, a similar trend has also been reported in [10, 7].

The popularity of obfuscation among malicious JavaScript code is caused by the following reasons. First, signature-based detection systems, e.g., anti-virus software, can be effectively evaded by obfuscation. For example, by applying encoding-based obfuscation on 100 known malicious samples, we demonstrated that all these obfuscated samples can successfully evade the detection of all popular anti-virus software listed in VirusTotal. Second, the dynamic features of JavaScript language, such as dynamic code generation and run-time evaluation, facilitate the creation of obfuscation routines. For instance, dynamic generation combined with a string manipulation process can easily generate an obfuscation function in JavaScript (e.g., [1]).

Many approaches have been proposed to detect malicious JavaScript code. Some of these approaches [9, 19, 17] focus on detecting obfuscated malicious JavaScript code, whereas the others [10, 23, 22, 12, 7] treat obfuscation as one of the features of malicious JavaScript code. To detect obfuscated malicious JavaScript code, these approaches either adopt machine learning techniques using features extracted from various representation levels (e.g., source code, lexical token, AST), or they perform runtime analysis. Machine learning based approaches can achieve high throughput, and they can also achieve high accuracy with a large set of features. However, they do not capture the essential characteristics of obfuscated malicious JavaScript code, thus easy to be evaded. On the other hand, runtime analysis can expose the deobfuscation behavior, but the performance penalty incurred by runtime analysis prevents it from being used in online or large-scale scenarios.

In this work, we propose JStill, a mostly static obfuscated malicious JavaScript detector. The design of JStill is inspired by the following observation: *obfuscated malicious JavaScript code has to be deobfuscated prior to fulfilling its malicious intent, and in JavaScript, the deobfuscation process has to invoke certain functions.* Based on this observation, JStill analyzes function invocations and identifies the ones that can be potentially involved in obfuscated malicious JavaScript code. Although the analysis is fully static, JStill also leverages certain runtime operations of a browser be-

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

cause static approaches are limited in inspecting JavaScript code. In runtime, JStill intercepts and examines the *suspicious function invocations before they are executed*. Note that JStill does not rely on the dynamic behavior of suspicious JavaScript code. The combination of static analysis and runtime inspection has been proposed in the literatures [21, 12, 23, 6]. In this work, JStill strikes the balance between these two components such that the requirements on both performance and accuracy can be met. Because JStill has a runtime component, it can not only detect obfuscated malicious JavaScript code, but also prevent detected malicious code from being executed in a browser. Moreover, we will show that the integration of JStill in a browser requires very few modifications and causes very small (negligible) performance impact.

In our evaluation of JStill, we use Alexa [5] top 50,000 web sites as the benign sample set (to test false positives), and over 30,000 obfuscated malicious JavaScript samples as malicious sample set (to test false negatives) . The evaluation results show that JStill has a small number of false positives and negligible false negatives. The performance overhead incurred by a prototype implementation of JStill only increases the average Web page loading time by 4.9%, which makes it an efficient scheme for detecting obfuscated malicious code.

1.1 Contributions

Our paper makes the following contributions:

- **Mostly static detection of obfuscated malicious code** We propose JStill, a lightweight, mostly static approach to detect obfuscated malicious JavaScript code, most of which can evade the detection of state-of-art anti-virus software. *JStill can not only detect, but also prevent the execution of obfuscated malicious JavaScript code on a user's Web browser.*
- **Function invocation based analysis** We present a function invocation based analysis technique that leverages the combination of static analysis and runtime inspection. Our analysis is based on three different aspects of function invocations so that it can effectively distinguish malicious obfuscated JavaScript code from benign code.
- **Evaluation** We implement a prototype of JStill in a commodity browser and evaluate its detection effectiveness. Our evaluation uses a large number of real-world samples, and its results demonstrate that JStill has a very low false positive rate and negligible false negative rate.
- **Realtime protection system** We also measure the performance overhead of JStill in terms of averaged increased webpage loading time. The results indicate the performance overhead incurred by JStill is very small, making it a practical intrusion prevention system (IPS).

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 introduces the background of JavaScript obfuscation. Section 3 provides an overview of JStill. Section 4 elaborates the design. Section 5 presents the evaluation results, followed by discussion in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2. BACKGROUND ON JAVASCRIPT OBFUSCATION

2.1 JavaScript Obfuscation

JavaScript obfuscation is to "make modifications to the program, changing the names of variables, functions, and members, making the program much harder to understand ... " [11]. Note that obfuscation is different from minification, which "removes the comments and unnecessary whitespace from a program" [11] to reduce the code size.

Both benign and malicious JavaScript code have been observed adopting obfuscation techniques; hence, obfuscation does not imply maliciousness. However, their purposes of obfuscation are different. Benign JavaScript code mainly leverages obfuscation to protect code privacy or intellectual property. This purpose requires obfuscated code to be human unreadable and without downgrading the execution performance. Malicious JavaScript code exploits obfuscation to hide its malicious intent; therefore, the obfuscated code aims to evade static inspection. Normally, execution performance is not a concern for attackers. In fact, attackers often apply multiple obfuscation to better hide their malicious intent. For example, a drive-by download attack in Figure 1 is concealed by two levels of obfuscation.

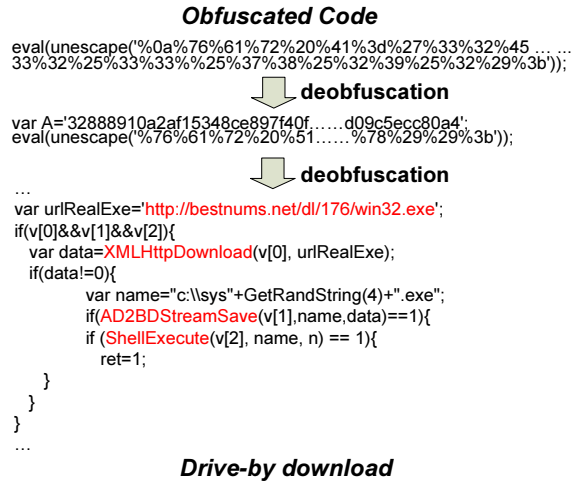

Figure 1: A real-world obfuscated drive-by download example

2.2 Obfuscation Techniques and Tools

The goal of this work is to detect obfuscated malicious JavaScript code. It is important to understand the obfuscation techniques adopted by malicious JavaScript code. To this end, we conducted a survey study. We randomly selected 100 (out of 510) known malicious JavaScript samples and manually examined the obfuscation techniques adopted by these samples. We summarized the techniques into the following five categories. Note that this categorization is by no means an exhaustive list, but it includes most of the basic building blocks that can be combined to generate more complicated obfuscation techniques as shown in [16].

Data Obfuscation, in which a variable or a constant is converted into the computational results of one or several variables or constants, e.g., string splitting and JavaScript keyword substitution. 47% of samples adopt this technique.

ASCII/Unicode/Hexadecimal Encoding, in which malicious JavaScript code is encoded into escaped ASCII char-

acters/unicode/hexadecimal representations. 32% of samples adopt this technique.

Customized Encoding Functions, in which malicious JavaScript code is obfuscated by a customized encoding function. Meanwhile, a corresponding customized decoding function is also attached to decode the obfuscated code at runtime. 23% of samples adopt this technique.

Standard Encryption and Decryption, in which the malicious JavaScript code is encrypted and decrypted for execution using standard cryptographic functions. 3% of samples adopt this technique.

Logical Structure Obfuscation, in which the execution paths of malicious JavaScript code is changed without affecting the original semantics, e.g., inserting some instructions that are independent of the functionality of the malicious code, adding/changing conditional branches, etc. 11% of samples adopt this technique.

Our results show that 71% of examined malicious samples employ obfuscation techniques (counting multiple obfuscation as one). Data Obfuscation appears to be the most popular technique. However, an in-depth analysis shows that most of the samples that adopt data obfuscation also adopt other obfuscation techniques such as encoding or encryption based obfuscation. In fact, 40% of the obfuscated malicious samples apply multiple obfuscation to further hide their malicious purposes. Figure 1 shows an example of multiple obfuscation. The first level is ASCII encoding obfuscation, and the second level is customized encoding function obfuscation.

More importantly, we notice two common operations among these obfuscation techniques. The first one is the recovery of the clear-text version of malicious code from an encoded string or other objects on the Web page. A similar observation has also been reported in [10]. Since JavaScript code is delivered as text, the code can be manipulated as text; meanwhile, any text can also be potentially executed as JavaScript code. This flexibility is leveraged extensively in obfuscated malicious JavaScript code because: 1) it is easier to manipulate or obfuscate text than code; 2) attackers can potentially hide code anywhere in a Web page as text. The second common operation is the execution of the recovered malicious code has to involve dynamic generation and runtime evaluation functions. However, the existence of these common operations does not always mean attacks because they are also heavily used in benign JavaScript code [25].

As part of the survey, we also investigated the obfuscation techniques adopted by the top 10 most popular JavaScript obfuscation tools[2](Appendix B). We notice that 7 out of 10 tools use both encoding/encryption based obfuscation and data obfuscation. The other 3 tools use only data obfuscation. To study the effectiveness of evading anti-virus software by data obfuscation and by encoding/encryption based obfuscation, we applied both types of obfuscation techniques on the 100 known malicious samples, respectively. We submitted the obfuscated samples to the 20 highest ranked anti-virus software [5]. The average detection rate on obfuscated samples using data obfuscation is 45.7% (Appendix D), whereas the detection rate on samples using encoding/encryption obfuscation is 0. Apparently, data obfuscation can not evade the detection of anti-virus software as effectively as encoding/encryption obfuscation. Therefore, our approach focuses on the detection of encoding/encryption based obfuscated malicious code.

2.3 Dynamic Generation and Runtime Evaluation

Dynamic generation (D-Gen) and runtime evaluation (R-Eval) functions play an important role in obfuscated malicious JavaScript code. D-Gen functions can generate JavaScript code from text in runtime, and R-Eval functions can evaluate a text string as code. These two features provide a means of transforming text to code in JavaScript, and this transformation is a typical operation in the deobfuscation of obfuscated malicious JavaScript code as we discussed above. Therefore, these two features are widely exploited in malicious JavaScript obfuscation. In the example of Figure 1, the obfuscation leverages the "`eval`" function to dynamically generate the deobfuscated code.

On the other hand, D-Gen and R-Eval functions are also commonly used features in benign JavaScript code [25]. In the case of conditional loading, an external JavaScript code is loaded using dynamic generation only when certain condition is met. This can avoid unnecessary bandwidth consumption. Another example is including runtime generated information in JavaScript code to increase the flexibility in programming. When JavaScript code contains information that can only be obtained during runtime from either user input or client-server interaction, it will leverage runtime evaluation. Therefore, the adoption of D-Gen and R-Eval functions does not always imply the existence of obfuscated malicious code.

3. OVERVIEW

This section gives an overview on the design of JStill.

3.1 Function Invocation Based Analysis

As we mentioned before, the basic observation in the design of JStill is that either the deobfuscation or the execution of obfuscated malicious JavaScript code has to involve function invocations. To leverage this observation, we first categorize functions in JavaScript into the following types: [1] 1) JavaScript native functions (e.g., `eval`), 2) JavaScript built-in functions (e.g., `unescape`, `string.fromCharCode`), 3) DOM methods (e.g., `document.write`, `window.setTimeout`) and 4) user-defined functions. Since both obfuscated malicious JavaScript code and benign JavaScript code invoke these four types of functions, the challenge here is how to distinguish function invocations in obfuscated malicious code from that in benign code.

To this end, JStill captures the essential difference between obfuscated malicious invocations and benign invocations from the following three aspects.

Function arguments. We notice that for some language-defined functions that are often used in deobfuscation, e.g., the D-Gen and R-Eval functions, malicious invocations of these functions often hide their arguments from the static perspective, e.g., using the output of another function as arguments. This is necessary for obfuscated malicious code because the arguments of these function invocations often contain part or all of the malicious code. Exposing these arguments will increase the chance of being detected by static inspections. For other language-defined functions used in deobfuscation, e.g., `unescape`, we noticed that the outputs of these functions are often used as or in the arguments of D-Gen and R-Eval functions. This is because in obfuscated

[1] the first three types of functions, which can also be collectively called language-defined functions.

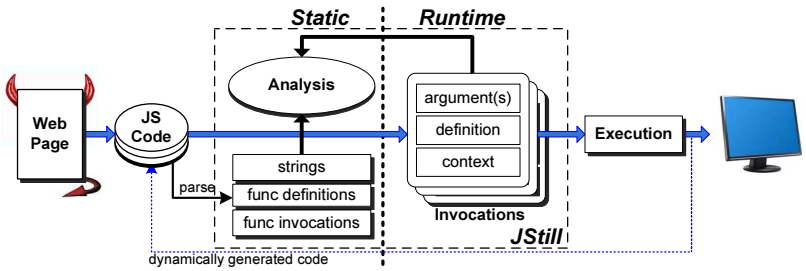

Figure 2: Overview of JStill

malicious JavaScript code, functions like `unescape` can decode the obfuscated string, which will later be generated or evaluated as code.

Function definition. In benign code, a user-defined function is normally first defined before it is invoked. However, in many cases of obfuscated malicious code, a malicious function's definition is either entirely or partially obfuscated in order to hide the semantics of the malicious code. Therefore, when the malicious function is invoked later, it would appear undefined from the static point of view, even though its definition has already been evaluated by a JavaScript engine. In addition to obfuscated malicious code, a coding bug can also cause invocations of undefined functions. The only difference here is that in a coding bug, the function is indeed undefined before its invocation.

Note that hiding definitions of user-defined functions is very rare in benign JavaScript code. As we mentioned before, the purpose of obfuscation for benign JavaScript code is mainly for intellectual property protection. Since JavaScript is delivered in source form, not as compiled machine code, its source code cannot be protected as much as in other languages. An instrumented browser can reveal the source JavaScript code to people who are interested, no matter what obfuscation is applied on that code. Therefore, rather than hiding the source code, most benign obfuscation focuses on reducing the interpretability of the source code to make it hard for human to understand the logic of the code. To this end, benign code favors randomization and substitution based obfuscation techniques. Another reason that benign code normally does not adopt dynamic generation based obfuscation is the concern with extra performance overhead, which is an important factor to evaluate the coding quality of a website.

The context of a function invocation. A context means what function is actually invoked. Based on our analysis of obfuscated malicious JavaScript code, we observed the disguise in invoking language-defined functions, in order to evade detection that leverages invocation patterns of language-defined functions.

Figure 3 lists two common disguise techniques. In part (a) of Figure 3, `eval()` function is assigned to an object `a`, which is later invoked as `eval()`. In part (b), the properties of a window object are traversed to look for the "document" keyword, which is assigned to object `b`, whereas object `c` actually contains the string "write" after the execution of "`unescape()`", so together the last statement actually invokes "`document.write()`". In these examples, static analysis may be able to trace back to the language-defined functions that are actually invoked, but it is neither reliable nor efficient. If the statement "`a = eval;`" in part

```
a = eval;
a('alert("test")');
```

(a)

```
for(b in window){
  if(b.length == 8){
    if(b[0] == 'd'){
      break;
    }
  }
}
c = unescape("%77%72%69%74%65");
this[b][c]('test'+'</br>');
```

(b)

Figure 3: Examples of disguised invocations of language-defined functions

(a) is escaped and evaluated in runtime using "`eval (unescape("%61%3D%65%76%61%6C")`", static analysis will not detect that "`eval`" has been assigned to "`a`" unless it has access to the runtime generated code.

3.2 System Overview

To detect obfuscated malicious JavaScript code, JStill uses static analysis to examine function invocations from the above three aspects: *function definition*, *content of arguments* and *context of invocation*. As illustrated in Figure 2, the static analysis first parses JavaScript code and logs information such as strings, function definitions and function invocations based on the parsing results. JStill leverages the static information about function definitions and invocations by comparing it with the runtime information about function definitions and invocations. This comparison can reveal what functions are statically undefined as well as what function definitions are hidden by obfuscation. The information about string is used by JStill in the analysis of hidden arguments, which will be discussed later.

Meanwhile, since static analysis itself is insufficient to discern coding bugs from obfuscated malicious code, or to identify disguised function invocations, JStill also leverages its runtime component to assist static analysis. In runtime, JStill hooks the invocations of selected language-defined functions in a browser. In this way, it can examine the suspicious arguments just before the execution since the arguments are in clear-format at this stage. JStill can also spot disguised invocations of these hooked functions. Because no matter what disguise is applied, the invocation will always be handled by the hooked functions such that JStill has an opportunity to check if the invoked function in the code is actually the hooked function.

Since JStill detects obfuscated malicious JavaScript code from three aspects, it consists of three detection criterions: 1) disguised invocations of language-defined functions, 2) obfuscated function definitions, 3) obfuscated malicious arguments. JStill raises an alarm if at least one criterion is

met. Note that the design of JStill does not rely on any unique specification in a browser's implementation. Therefore, JStill can be implemented compatibly in any Web browser.

4. THE DESIGN OF JSTILL

In this section, we explicate the design of JStill, particularly the technical challenges and their solutions in enforcing the three detection criterions.

4.1 Identification of Disguised Function Invocations

To identify a disguised function invocation, two pieces of information are necessary: 1) what function is actually invoked in an invocation; 2) what function appears to be invoked in an invocation.

To gather the information about what function is actually invoked, static approaches, such as tracing back to the actual function being invoked, are either unreliable or infeasible. Therefore, JStill leverages runtime inspection to identify the function that is actually invoked. More specifically, JStill hooks the implementation of language-defined functions that are mostly likely to be disguised in obfuscated malicious JavaScript code. These functions include but not limited to D-Gen and R-Eval functions (e.g., the functions disguised in Figure 3,"`eval`" and "`document.write`"), and functions that are commonly used in string manipulations (e.g., "`unescape`", etc.). These functions are mostly likely to be disguised because their prevalence in obfuscated malicious code makes them (part of) the widely used detection signatures in static inspections. When one of these hooked functions is invoked by a function invocation, JStill can identify the hooked function as the actually invoked function.

However, hooking a function's implementation cannot reveal what function appears to be invoked. Many of these functions (e.g., DOM based functions) are not implemented within the JavaScript engine; the invocations of these function are actually wrapped by a component (e.g., XPConnect in Firefox) that allows JavaScript code to invoke these functions without revealing the function name in the invocation (e.g., "`a`" in Figure 3(a)).

To address this issue and obtain the information about what function appears to be invoked, JStill marks all the statically identified invocations of a hooked function (not only from source code, but also from dynamically generated code). As a result, when a hooked function is invoked during runtime, JStill can check if this invocation has been marked; any unmarked invocation in this case indicates this invocation is disguised from the static perspective.

The marking scheme must cover all the statically identifiable invocations of hooked functions to eliminate false positive in disguised invocation detection, but identification of function invocations in JavaScript is not a trivial task. Since function in JavaScript is merely a special type of object, it can be assigned as variables, stored as properties in other objects or as elements in arrays. In other words, a function definition can be passed as a value from one object to another object. For example, Figure 4 lists different means by which "`function addition()`" is passed to various objects (or properties) and gets invoked. In all four cases, the last statement actually invokes "`addition(2, 3);`". In invocation 1, function "`addition`" is passed to an array element, e.g., "`arr[0]`". In invocation 2, it is passed to an object's (including other function object's) property, e.g., "`obj.func2`".

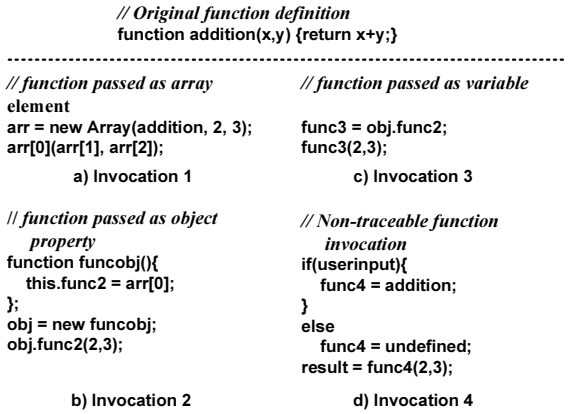

Figure 4: Example of legitimate function invocations in JavaScript

After that, it is passed to a variable and gets invoked by a variable's name, e.g., "`func3`" in both invocation 3 and 4.

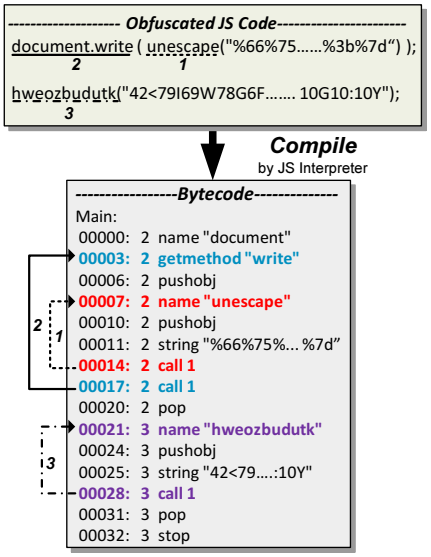

Figure 5: Identify function invocations via bytecode

To identify all the function invocations despite the flexibility in the syntax of JavaScript, necessary syntactic information needs to be parsed from the source code. To this end, JStill leverages the bytecode that is compiled from source code, an example of which is shown in Figure 5. From bytecode in Figure 5, it is clear that there exist three function invocations (the three bytecode "call" marked by different colors). To leverage the bytecode to identify function invocations, JStill needs to understand the structured syntactic information offered by bytecode. The information is organized as a set of 3-tuples. As illustrated in Figure 5, each line of bytecode is a 3-tuple that represents a sequence number, a line number and a bytecode instruction, respectively.

When marking a function invocation for runtime inspection, JStill needs to make sure that the mark cannot be bypassed by malicious code. Meanwhile, it also tries to avoid modification on the source code to prevent unwanted impact on the runtime behavior of the code. As a result, JStill

actually marks invocations by logging the locations of these invocation instructions. In this way, when a hooked function is actually invoked, JStill can determine whether the invocation is disguised by checking if the location of instruction for this invocation is marked. An unmarked bytecode will indicate a disguised invocation of a hooked function.

The same approach however cannot be applied on user-defined functions because these functions are not implemented in the browser and hence cannot be hooked. To identify disguised invocations of user-defined functions, JStill leverages the object hierarchy in JavaScript. Every user-defined function is a property of its parent object. When a user-defined function is invoked, JStill can identify what property this invocation actually uses. If the property's name and the caller in the invocation do not match, it means the invocation is disguised.

Note that bytecode, as an intermediate interpretation of JavaScript source code, has different forms in various browsers (e.g., Firefox, Safari and IE). However, it is not indispensable to a browser's implementation (e.g., Chrome's JavaScript engine V8 escapes bytecode) or to JStill. The reason to use bytecode generated by a JavaScript engine rather than parsing the source code by JStill itself is that commodity JavaScript engines such as SpiderMonkey are highly optimized and robust facing malformed JavaScript code.

4.2 Detection of Obfuscated Function Definitions

As we discussed in the overview section, the second aspect in JStill's analysis of invocation is function definition. Since an obfuscated definition of a user-defined function is an indication of obfuscated malicious JavaScript code, we will discuss how to detect obfuscated function definition in this section.

A function definition can be obfuscated by either hiding the entire function definition or a part of the function body. When the entire function definition is hidden, the definition cannot be observed in the process of parsing the source code. Hence, an invocation of this function would appear statically undefined. However, a coding bug may appear the same way. Therefore, to provide a more accurate detection of obfuscated function definition, JStill checks every invocation to see whether the invoked function is actually defined or not. If the actually invoked function is defined only in runtime and the definition is hidden from static perspectives, it is an obfuscated function definition.

To examine whether a function has been defined or not, JStill checks all the function definitions it logs in parsing JavaScript code (both source code and dynamically generated JavaScript code). In this process, JStill uses both function names and the object hierarchy to match a function definition, since function name based definition match can cause inaccuracy due to different JavaScript contexts. For example, functions with the same name can be defined within different objects. Therefore, to accurately match a function definition, information about where the function is defined also needs to be checked. Such information can be obtained by checking the object hierarchy of a function definition. Therefore, object hierarchy is also logged together with function definition during the course of parsing.

To identify obfuscated function definitions, JStill checks every function definition identified in runtime-generated code. If a function definition is generated from obfuscated arguments (the detection of which will be discussed in the next Section) of D-Gen and R-Eval functions, it is an obfuscated definition.

A function definition can also be partially obfuscated, i.e., only part of the function body is hidden. In this case, the function must contain code that is dynamically generated using obfuscated arguments. Therefore, partially obfuscated function definition can be identified via the detection of obfuscated arguments within a function body. Specifically, JStill marks the invocations of D-Gen and R-Eval functions within a function body in parsing. When the marked invocations are detected as containing obfuscated arguments in JStill, partial obfuscation in the function body can also be detected.

Another practical issue is that there exist cases in which the function definition of an invocation cannot be determined statically. For example, in invocation 4 of Figure 4, the value of the variable "`userinput`" cannot be determined statically, so the actual definition of "`func4`" remains unknown from the static viewpoint. To solve this issue, JStill leverages the result from identification of disguised invocations such that it can determine which function definition is actually invoked before checking definition obfuscation.

4.3 Detection of Obfuscated Malicious Arguments in D-Gen and R-Eval Functions

D-Gen and R-Eval functions used in obfuscated malicious code often obfuscate their arguments, as we mentioned in Overview. Since these functions can transform text to JavaScript code, their arguments are hidden in the form of the outputs of string manipulation functions. These functions can be either language-defined or user-defined. In the example of Figure 5, the argument of `document.write` is the output of `unescape`. Moreover, the trace from the output of a string manipulation function to the arguments of D-Gen and R-Eval functions can be obfuscated as well. For the code in Figure 5, an attacker can change the first statement to "`document.write(s);`", where "`s`" is a string that actually equals to "`unescape("%66...%7d")`" except this equivalence is disguised by other statements crafted by attackers.

Hooking D-Gen and R-Eval functions provides an opportunity to examine the content of arguments, but the content itself does not shed any light on whether it has been obfuscated. Besides, patterns that show a resemblance to the obfuscated malicious code in Figure 5 have been observed in benign JavaScript code as well , e.g., "`document.write('<scr' + 'ipt src="' + urlStart + ".2mdn.net/" + iframeScript-File + '">');`". In view of this, detection of obfuscated arguments in D-Gen and R-Eval functions is a challenging problem.

To solve the problem, JStill introduces an obfuscated malicious argument (OMA) metric for all the arguments of dynamic generation and runtime evaluation functions. This metric indicates the possibility that an argument is used in obfuscated malicious code. The main purpose of applying obfuscation on malicious arguments is to hide the content of the malicious arguments; hence, the malicious arguments, or most of the arguments must not be observed from the source code. In benign code, the arguments do not need to be hidden, but often need to be dynamically assembled (or concatenated), since some parts of the benign arguments depend on user input or environment variables. In other words, in benign JavaScript code, most (if not all) of the content of the arguments can be found in the Web page (including URLs). Based on this observation, JStill defines the OMA metric as *the percentage of an argument that can be found*

in the Web page. For arguments with a low value on this metric, it is highly likely that they are obfuscated malicious code.

Since only the arguments that generate JavaScript code are potentially involved with obfuscated malicious code, other arguments can be excluded from examination to improve performance. For example, JStill rules out the invocations of D-Gen and R-Eval functions which create new HTML elements that are neither a script tag nor containing any event handler, because these arguments will not introduce new JavaScript code. Besides, arguments that create script tags used for dynamic inclusions (e.g., `<script src="a.js">`) are also excluded from this examination, because the dynamically included code will be analyzed by JStill later.

To calculate the metric of an argument, e.g., a string, JStill logs all the string variables from the parsed source code and the values of some environment variables that are commonly used in benign JavaScript code, such as window.location.href, navigator.userAgent, element IDs, etc. The details on the calculation of the metric will be explained in Section 4.4.

JavaScript provides many functions that can be used for D-Gen and R-Eval. Appendix C lists the functions that JStill hooks in Firefox. We realize that this list is browser specific (e.g., Firefox in this work), but the design of JStill is not exclusive to a certain browser. We will further discuss the discrepancy caused by different browsers in Section 6.

4.4 Obfuscated Malicious Argument Metric

OMA metric measures the possibility of an argument being used in obfuscated malicious JavaScript code. Given a set of strings and the content of an argument, the metric is calculated as the largest percentage of the argument's content that can be found as the combination of the strings or substrings from the set.

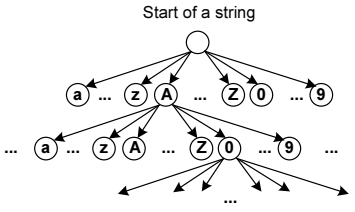

Figure 6: A Search Tree for Basic Strings

In benign JavaScript code, not only the strings, but also their substrings may be used in the composition of benign arguments. Therefore, the logged strings in the set are first divided into basic strings, which are substrings consisting of consecutive letters and numbers with a minimal length of 2 characters. This is because benign code mostly uses substrings that are divided by symbol separators, such as query content divided by questions mark in a URL, or cookie id after an equal sign.

Given a set of basic strings (let the size of the set be p, the average length of the strings in the set be m), the first calculation step is to find which basic strings are substrings of the argument (with size n). A brute-force algorithm has a complexity of $O(pmn)$. This will cause a significant performance penalty when p and n are large numbers. In fact, p is usually large due to the large number of strings defined in JavaScript code.

To avoid the performance penalty, we propose a search tree based algorithm. As illustrated in Figure 6, each internal node can have at most 62 children, which are mapped to the characters set (a-z,A-Z,0-9). This tree has p leaf nodes and an average depth of m. Based on this tree, JStill can accomplish the first step using the Substring Identification Algorithm (Appendix A.1)

Given the set of matched basic strings, the next step is to find a subset so that the combination of strings in this subset matches the largest percentage of the argument. This problem reduces to the knapsack problem, which is NP-Complete, so we propose an approximate, the Metric Calculation Algorithm (Appendix A.2) that leverages the following greedy heuristic: always using the longest matched and non-overlapping substring. In practice, this algorithm works very well, because the overlapped substrings are not very common.

The time complexity of the Substring Identification Algorithm (Appendix A.1) is O(mn), and the time complexity of the Metric Calculation Algorithm (Appendix A.2) is O(qn), where q is the number of matched substring set output by the Substring Identification Algorithm.

4.5 Whitelisting

As the most popular client-side language in Web development, JavaScript has many libraries and widgets. Most of these libraries are frequently used in many websites and are known to be benign, e.g., JQuery. Therefore, it is only a waste of resource to examine these known benign libraries in JStill.

To save the resource and to improve performance, we propose a whitelist mechanism in JStill. For a known JavaScript library that is often included as an external file, its hash value is computed and stored in JStill. During the examination, the hash value of every fetched external JavaScript file will be compared with the stored hash values. A match in the comparison will exempt the external file from further inspection. The same whitelist scheme can also be applied to web pages when JStill resides in a Web proxy and inspects all the incoming Web contents going through the proxy. When an http request hits the cache in the proxy, the requested Web page must have been examined by JStill; hence, JStill need not inspect the page again.

4.6 Prevention of Malicious Obfuscated JavaScript Code

The runtime inspection component can not only be used in the detection of obfuscated malicious JavaScript code, but also in prevention of the execution of detected malicious code in a browser.

```
<script>
…
flag = mal_ob();/* ob() is malicious obfuscated function */
if (typeof flag == "function") {
   benign();   /* the following is benign code */
   …
}
…
</script>
```

Figure 7: An example of disrupting benign execution

For obfuscated malicious code that uses invocations of D-Gen and R-Eval functions, since JStill intercepts the arguments of the invocations, it can replace the malicious content with NULL and continue the execution. For obfuscated malicious code that uses user-defined functions, the same ap-

proach may disrupt the normal execution of the benign JavaScript code in some cases. Since the basic interpretation unit in most JavaScript engines is a code segment enclosed by "<script>" tags, disabling a detected malicious function will lead to the following benign code in the segment being skipped. For example, as illustrated in Figure 7, if the detected obfuscated function invocation "`mal_ob()`" is disabled or commented out, the rest of the benign JavaScript code will not be executed because the type of "flag" will be "undefined".

JStill tries to prevent the malicious code from being executed while keeping the execution of benign JavaScript intact. Therefore, JStill substitutes the invocation of a detected malicious user-defined function with an invocation of a NULL function, which does nothing except returning a NULL. In this way, when a user's browser renders the web pages, these NULL function invocations will reduce the possibility of interrupting the execution of benign JavaScript code.

One concern regarding this substitution-based prevention scheme is that the false positives in obfuscation detection may cause loss of functionalities in the Web pages. However, based on our evaluation, which will be described in Section 5, normally this is not a big issue even a false positive occurs.

5. EVALUATION

In this section, based on a prototype implementation, we evaluate JStill in terms of (1) detection effectiveness and (2) performance overhead. The prototype of JStill is implemented in Firefox (version 3.6), which uses a rendering engine Gecko (1.9.2) and a JavaScript engine SpiderMonkey (version 1.8). The implementation adds 1.1 KLOC into the source code of Firefox. We also automate the process of rendering a Web page from the instrumented Firefox using Python scripts such that the browser can check against either a list of URLs or a directory of offline Web pages.

5.1 Evaluation Setting

The prototype of JStill is tested in Ubuntu 8.0, which runs a Pentium 4 3.4 GHz single-core CPU, 1 GB RAM, 160 GB 7200 RPM hard drive and 100 Mbps ethernet interface.

Sample Collection We collect both benign and malicious samples from the real world. The benign sample set consists of Web pages crawled from Alexa [5] top 50,000 websites. The malicious samples are collected from VirusTotal (flagged by ≥ 5 AV vendors). There are two sets of malicious samples. The first set contains 2,327 samples, among which 1,499 samples include obfuscated malicious JavaScript code (identified by manual examination). The second set contains 10,501 samples. Since these samples have already been detected by AV vendors, to better evaluate the effectiveness of JStill in detecting obfuscated malicious code, we apply 3 JavaScript obfuscation tools on each sample in the second set. This process generates another 31,505 obfuscated malicious samples.

Table 1: False Positives and False Negatives in Malicious Obfuscation Detection

Obfuscation Metric Threshold	FP	FN
1	2.19%	0%
0.9	1.90%	0%
0.8	1.89%	0.13%
0.7	1.75%	0.53%

5.2 Detection Effectiveness

Table 1 shows the overall detection accuracy of JStill in the evaluation using both benign and malicious sample sets. Each row in Table 1 corresponds to a different obfuscated malicious argument (OMA) metric threshold used in evaluation. Note that OMA metric is the only adjustable detection criterion in JStill, the other two criterions are obfuscated function definitions and disguised invocations of language-defined functions. The purpose of choosing different values is to understand how detection accuracy (i.e., FP and FN)is affected by the threshold of OMA metric.

One insight from the results in Table 1 is that the OMA threshold leads to a trade-off between false positive rate and false negative rate. When the threshold is low (e.g., 0.5), the false positive rate is also low (1.63%). This is because some arguments of benign invocations of D-Gen and R-Eval functions contain strings that cannot be found in Web pages; thus, these arguments have relatively low OMA metric values. However, when JStill adopts a low threshold on obfuscation metric, these arguments may not cause false positives, hence the false positive rate is low.

On the other hand, when the threshold is high (e.g., 1.0), the false negative rate becomes low (0%). A high threshold means that an argument can be considered as benign only when a large portion of the argument can be found in the Web page. Attackers can increase the OMA metric of some malicious arguments (e.g., by only obfuscating part of the malicious content and leaving the rest of the arguments in plaintext) and cause false negatives by surpassing the threshold. However, this becomes very hard when the threshold is set high, because passing a high threshold requires most of the arguments not being obfuscated, in which case the chance of malicious code being detected by signature-based approaches also increases. Note that JStill is not designed to replace the signature-based schemes, but instead they are complementary to each other.

Table 2: Composition of False Positives

Cause of False Positives	%
Obfuscated arguments of D-Gen and R-Eval functions	95.89%
Disguised invocations of language-defined function	0.46%
Obfuscated function definitions	3.65%

False Positives Table 2 lists the false positives incurred on each detection criterion of JStill. Most of the false positives are caused by obfuscated arguments of D-Gen and R-Eval functions. There are two main causes of the non-negligible false positives. The first cause is that information generated at runtime(e.g., random number, user inputs) takes a large portion in the arguments of the R-Eval functions. Given the fact that the arguments of R-Eval functions are JavaScript code and a large portion of the code cannot be observed in any static perspective, this case is very similar to that of obfuscated malicious code.

The second cause is that some benign Web pages actually adopt encoding-based obfuscation on some parts of their JavaScript code. One example is the Web page retrieved by the URL "www.360buy.com". In this Web page, the argument of an invocation to "`eval`" is encoded using a customized encoding function. Meanwhile, a decoding function is also observed as part of the code. After decoding, the execution of "`eval`" evaluates a large body of JavaScript code, which is actually a JQuery library. Given the open-source nature of JQuery, the purpose of this obfuscation is not clear. Indeed,

when benign code adopts the same obfuscation techniques as malicious code, the problem of differentiating one from the other is probably undecidable. We believe this problem may only be solved by observing the runtime behavior, which, however, is not an efficient approach to be deployed in any large-scale or realtime scenarios, not to mention the challenge in traversing all execution paths.

One concern regarding the false positives is the possibility of interrupting user's browsing experience. However, in reality this normally is not a big concern. This is because: 1) the prevention scheme in JStill does not hinder the execution of the rest of the code; 2) Most of the false positives only affect a single function invocation in a Web page. Considering the popularity of tools such as NoScript, nullifying a single JavaScript function invocation would probably not affect user's browsing experience.

It is also worth noting that we implicitly assumed that none of the 50,000 Web pages as well as their linked .js files is malicious in our evaluation. To verify whether it is the case, we will have to resort to a dynamic analysis approach (e.g., [8]). However, if some of these Websites are indeed malicious, the false positive rate of JStill will only be lower.

False Negative The analysis of JStill's false negative rate is based on the examination of both obfuscated malicious sample sets. The overall false negative rate is listed in the third column in Table 1.

Since false negative rate is related to the threshold of the OMA metric, when the threshold is high (e.g., 1), the overall false negative is 0. That is to say JStill can detect all the malicious obfuscated JavaScript code in our malicious sample set when the threshold is set to 1. When the threshold is low (e.g., 0.7), false negatives start to happen in the first set of malicious samples. These false negatives are incurred on the criterion using the OMA metric. Most of the samples causing false negatives obfuscate only a part of the malicious arguments in D-Gen and R-Eval invocations. As we discussed before, this will also increase the chance of being detected by signature-based approaches. In fact, these samples are detected by multiple AV vendors. There is no false negatives in the second set even when the threshold of OMA metric is low. This means JStill can effectively detect obfuscated malicious code that is generated by JavaScript obfuscation tools.

5.3 Comparison with Other Techniques

Although there are many works on malicious JavaScript code detection, these works either detect general malicious JavaScript (e.g., Prophiler [7],JSAND [10]) or focus on specific malicious JavaScript (e.g., Nozzle [22], Zozzle [12]). Therefore, they are not comparable with JStill. However, several other works also focus on the detection of obfuscated (malicious) JavaScript code, so in Table 3 we compare them with JStill in terms of detection effectiveness. Further discussions on the difference between JStill and these schemes can be found in the related work section.

Table 3: Comparison with Existing Approaches on Detection of Obfuscated (Malicious) JavaScript Code

Approach	FP	FN
NoFus [17]	1%	5%
[18]	12.13%	3.84%
JStill	1.75%	0.53%

5.4 Performance Overhead

In our evaluation, performance overhead is measured in terms of *the average increased loading time* for Web pages. To factor out the random fluctuations in network latency, JStill automatically visits the same website 20 times. Meanwhile, the option of caching visited Web pages is also disabled in the instrumented browser to make sure the Web pages are retrieved from the Web server instead of being loaded from the local cache. To calculate the overhead in loading time, the same evaluation is preformed again using an non-instrumented Firefox, and the difference between the results from two evaluations is the performance overhead. Figure 8 lists the average performance overhead in loading time for Alexa top 20 websites. The overall average performance overhead is 4.9%, which makes JStill a practical realtime detection tool.

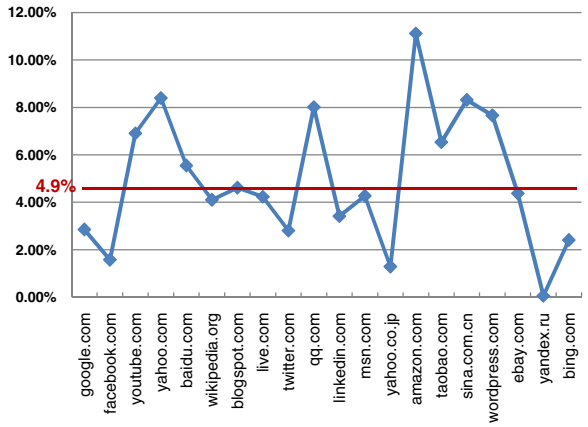

Figure 8: Average Performance Overhead for Top 20 Websites

6. DISCUSSION AND FUTURE WORK

Minification v.s. Obfuscation Minification in JavaScript works by removing the comments and unnecessary whitespace from the code. Its purpose is to reduce the size of code, not to hide the code. Although the readability of minified code may be reduced as a result, it is different from obfuscation. As discussed in [11], some obfuscation tools may include minification as one of the steps, but they in addition modify the JavaScript code so that the logic of the code can be concealed.

In this work, we focus on obfuscation, not minification, because minification will not help attackers hide the malicious JavaScript code from inspection.

Discrepancy in Browsers Different Web browsers may render Web pages differently, parse and interpret JavaScript code differently according to their implementations. One practical concern is that the discrepancy in browser implementations will affect the detection effectiveness of JStill. For example, scripted browser detection and conditional comments can cause different scripts being executed in different Web browsers. Since JStill is based on static analysis, even if a part of script is not executed because of scripted browser detection, it is still parsed and examined by JStill. For conditional comments, we acknowledge that the IE-targeted scripts cannot be observed by an implementation of JStill on Firefox. However, as we mentioned, the design of JStill does not rely on any unique feature from Firefox. JSTILL can also be implemented on other Web browsers. In this

work, we choose Firefox as the platform because it is the most popular open source browser [3, 4].

Other Evading Techniques JStill is designed to detect obfuscated malicious JavaScript code that hides the malicious code from static analysis, because this is the most effective way of evading static inspection and the most prevalent choice among the observed malicious JavaScript code. However, we do realize that there exist other forms of obfuscation, e.g., variable and function name randomization or substitution, etc. One common characteristic among these simple forms of obfuscation is that the semantics of the malicious code is not hidden. Although they may be able to evade some signature-based detection systems, they cannot evade an effective static detection tool that captures the unchangeable parts in malicious code. These unchangeable parts include exploited vulnerabilities (e.g., an ActiveX object), language-defined functions (e.g., "eval") or the location of a malicious payload (e.g., "http://foo.com/malware.exe"). Therefore, in this work, we focus on the obfuscation that actually hides the malicious code (or part of the code), and we believe JStill is a nice complement to other detection approaches, e.g., signature-based approaches and dynamic analysis based schemes.

Other Scripting Languages Malicious obfuscation is also seen in other scripting languages (e.g., VBScript, JScript) that are supported by specific browsers (e.g., IE). Some of these languages such as JScript are derived from the same ECMAScript standard as JavaScript. Therefore, the design principle of JStill can be applied to detect malicious obfuscated code in these languages. For other languages such as VBScript, despite the difference in syntax and language features, we believe the general idea of JStill, such as function invocation based analysis and the combination of static and runtime analysis can still shed some light in the new context.

7. RELATED WORK

JStill is the only approach that leverages both bytecode representation and runtime of JavaScript code. Figure 9 shows how JStill differs from existing approaches based on the representation level of JavaScript code used in each approach. It also illustrates the objective of a related approach, focusing on detection obfuscation or malicious JavaScript.

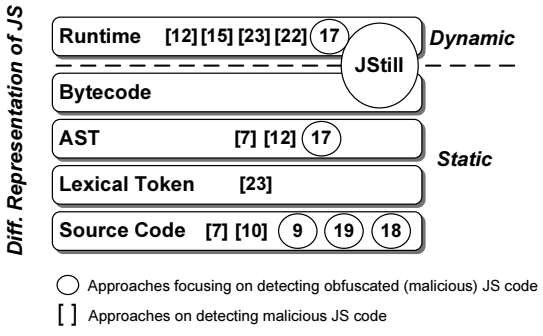

Figure 9: Comparison of JStill with other approaches

Obfuscated (Malicious) Javascript Detection. Figure 9 lists the four approaches that focus on the detection of obfuscated (malicious) JavaScript code. The one closest to JStill is NoFus [17]. NoFus is a follow-up of Zozzle [12], and it also leverages an AST based static classifier. NoFus detects if a piece of JavaScript code has been obfuscated for any purpose, and it also distinguishes benign obfuscated JavaScript and malicious JavaScript. NoFuS achieves a false positive rate of 1% and a false negative rate of 5%. JStill is different from NoFuS in the following aspects: 1) instead of AST representation, JStill leverages the bytecode representation, which contains more semantic information. 2) JStill focuses on detecting obfuscated malicious JavaScript code, instead of detecting obfuscation. 3) JStill achieves a comparable false positive rate, and a much lower false negative rate.

Likarish *et al.* [19] also leveraged classification techniques to detect obfuscated malicious JavaScript code. The features they applied on their classifier include length of the code, number of strings in the code, percentage of whitespace, etc. Choi *et al.* [9] proposed an approach to detecting the obfuscated strings in malicious JavaScript code based on the characteristics of obfuscated string objects, such as excessive usage of specific characters and excessive length of the string. Kim *et al.* [18] proposed an entropy-based system to detect obfuscated malicious JavaScript. JStill is different from these three approaches in: 1) JStill leverages function invocation based analysis, which captures the difference between obfuscated malicious JavaScript code and other JavaScript code. 2) Both false positives and false negatives rates of JStill is lower than those reported.

Malicious JavaScript Detection. Figure 9 also lists six approaches for detection of malicious JavaScript code. Curtsinger *et al.* [12] proposed Zozzle, a JavaScript malware detection system that also combines both static and dynamic analysis. As the precedent of NoFus [17], Zozzle is also based on machine learning using features extracted from AST of JavaScript code. Because Zozzle is trained using samples collected by Nozzle [22], it is more effective in classifying malware with heap spray and shellcode instead of obfuscated malicious code. There exist similarities between the purpose of the runtime component in JStill and in their work, since both components provide view of dynamically generated code. However, JStill also compares the information parsed from dynamically generated code with the information obtained from static analysis. Ratanaworabhan *et al.* [22] proposed a runtime approach to detecting heap spray attacks. Their approach monitors a browser's heap to identify structured x86 code (e.g., NOPs) that are often exploited by a heap spray attack. In JStill, the runtime inspection is lightweight and only focuses on accessing dynamically generated code and identifying JavaScript invocations. Cova *et al.* [10] proposed a detection scheme for malicious JavaScript code. Their scheme detects the obfuscated malicious JavaScript code by extracting features such as ratio of string definitions and string uses, length of dynamic code generation during the execution. These features, however, can be evaded by obfuscation techniques that manifest differently. Instead, JStill relies on more fundamental characteristics of obfuscation that cannot be easily evaded by "crafted" obfuscation. Hallaraker *et al.* [15] proposed an auditing system to examine the execution of JavaScript code on the client-side. To detect malicious JavaScript code, the audited code is compared with policies that specify the suspicious activities. Egele *et al.* [13] proposed an approach to detecting drive-by download by identifying JavaScript code that contains shellcode through x86 instruction emulation. Rieck *et al.* [23] proposed a system for detection and prevention of drive-by-download attacks using the combination of static and dynamic analysis. The static analysis in their work focuses on features extracted from lexical tokens, and

the dynamic analysis leverages a JavaScript sandbox such that it can reveal the runtime behavior of JavaScript code. In JStill, the analysis is mostly static, and the purpose of a runtime component is to provide access to dynamically generated and evaluated code.

8. CONCLUSIONS

This paper presents JStill, a mostly static approach to detect obfuscated malicious JavaScript code. JStill focuses on three aspects of function invocation analysis to provides efficient and effective detection and prevention of obfuscated malicious JavaScript code. It leverages the comparison of information obtained from both static analysis and runtime inspection. An evaluation has demonstrated the detection effectiveness of JStill as well as the low performance overhead. We see JStill as a good and practical complementary approach to existing signature-based detection systems. We also believe the design of JStill can shed some light on other obfuscation detection problems.

9. ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments and suggestions. This work was partially supported by NSF CAREER 0643906.

10. REFERENCES

[1] Online JavaScript Obfuscator. `http://www.daftlogic.com/projects-online-javascript-obfuscator.htm`.

[2] Javascript obfuscators review. `http://javascript-reference.info/javascript-obfuscators-review.htm`, 2006.

[3] Browser statistics. `http://www.w3schools.com/browsers/browsers_stats.asp`, 2011.

[4] Browser stats. `http://upsdell.com/BrowserNews/stat.htm`, 2011.

[5] Alexa. Alexa top global sites. `http://www.alexa.com/topsites`, 2010.

[6] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 387–401.

[7] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web* (New York, NY, USA, 2011), WWW '11, ACM, pp. 197–206.

[8] Chenette, S. ToorConX: The Ultimate Deobfuscator. `http://securitylabs.websense.com/content/Blogs/3198.aspx#`, 2010.

[9] Choi, Y., Kim, T., Choi, S., and Lee, C. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *Proceedings of the 1st International Conference on Future Generation Information Technology* (Berlin, Heidelberg, 2009), FGIT '09, Springer-Verlag, pp. 160–172.

[10] Cova, M., Kruegel, C., and Vigna, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 281–290.

[11] Crockford, D. Minification v obfuscation. `http://yuiblog.com/blog/2006/03/06/minification-v-obfuscation/`, 2006.

[12] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th conference on USENIX security symposium* (2011), USENIX Association.

[13] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2009), DIMVA '09, Springer-Verlag, pp. 88–106.

[14] Fossi, M., Johnson, E., and Mack, T. Symantec global internet security threat report. Tech. rep., Symantec, 2009.

[15] Hallaraker, O., and Vigna, G. Detecting malicious javascript code in mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems* (Washington, DC, USA, 2005), ICECCS '05, IEEE Computer Society, pp. 85–94.

[16] Howard, F. Malware with your mocha? obfuscation and anti emulation tricks in malicious javascript. Tech. rep., Sophos, 2010.

[17] Kaplan, S., Livshits, B., Zorn, B., Siefert, C., and Curtsinger, C. "nofus: Automatically detecting" + string.fromcharcode(32) +"obfuscated ".tolowercase() + "javascript code". Tech. rep., Microsoft Research, 2011.

[18] Kim, B.-I., Im, C.-T., and Jung, H.-C. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. In *International Journal of Advanced Science and Technology* (2011).

[19] Likarish, P., Jung, E., and Jo, I. Obfuscated malicious javascript detection using classification techniques. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software* (Oct 2009), MALWARE '09, pp. 47–54.

[20] Percoco, N. J. Global security report 2010 analysis of investigations and penetration tests. Tech. rep., SpiderLabs, 2010.

[21] Rabek, J. C., Khazan, R. I., Lewandowski, S. M., and Cunningham, R. K. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM workshop on Rapid malcode* (New York, NY, USA, 2003), WORM '03, ACM, pp. 76–82.

[22] Ratanaworabhan, P., Livshits, B., and Zorn, B. Nozzle: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), USENIX Association, pp. 169–186.

[23] Rieck, K., Krueger, T., and Dewald, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 31–39.

[24] Symantec. ActiveX file overwrite delete vulnerabilities. `http://www.symantec.com/connect/blogs/activex-file-overwritedelete-vulnerabilities-continued`, 2008.

[25] Yue, C., and Wang, H. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), WWW '09, ACM, pp. 961–970.

APPENDIX

A. ALGORITHMS

Algorithm 1 Substring Identification

Input: Basic strings set $S = s_1, ..., s_p$ in tree S_T, Argument string arg
Output: Subset of basic strings C

$C \leftarrow \emptyset$
$T \leftarrow S_T$
$i \leftarrow 0$
searchtree(T, i)
while $i < length(arg)$ **do**
  search level 1 of tree T for key $arg[i]$
  if key $arg[i]$ is found on node j **then**
    if node j is leaf **then**
      $C \leftarrow s$ ($s \in S$ $s :=$ path from root to j)
    else
      $T_j :=$ subtree from node j
      searchtree(T_j, $i \leftarrow i + 1$)
    end if
  end if
  $i \leftarrow i + 1$
end while

Algorithm 2 Metric Calculation

Input: Substrings set $S = s_1, ..., s_q$, argument string arg
Output: max percentage max_p

$max_p \leftarrow 0$
while $S \neq \emptyset$ **do**
  find the longest substring s_i in S
  $max_p \leftarrow max_p + length(s_i)/length(arg)$
  $S \leftarrow S - \{s_i\}$
end while

B. JAVASCRIPT OBFUSCATION TOOLS

Table 4: JavaScript Obfuscation Tools

Tools	Techniques
Thicket	D, A, S
Jasob	D
JS Obfuscator	D, A
Stunnix	D, A
JCE Pro	D, A
ScrEnc	D, A, C
Shane	D, A
Dean	D, A
Jammer	D
JSCrunch Pro	D

D:Data Obfuscation
A:ASCII/Unicode/Hexadecimal encoding
C:Customized Encoding Functions
S:Standard Encryption and Decryption
∗Encoding/encryption based obfuscation includes A,C,S

C. D-GEN AND R-EVAL FUNCTION HOOKED BY JSTILL

Table 5: D-Gen and R-Eval Function hooked by JStill

Function Name	Description
document.write	dynamic code generation
document.writeln	dynamic code generation
window.setTimeout	evaluate the 1st argument as code
window.setInterval	evaluate the 1st argument as code
eval	evaluate the argument as code

D. EVADING EFFECTIVENESS

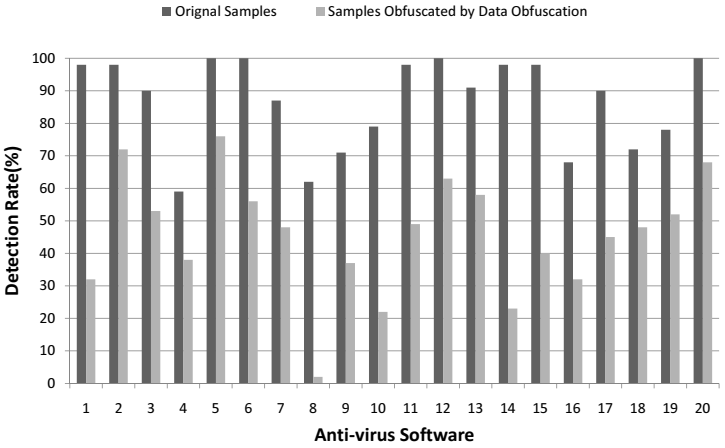

Figure 10: The Detection Rate of 20 Anti-Virus Software on Samples Obfuscated by Data Obfuscation