Protecting JavaScript Apps from Code Analysis

Tobias Groß
tobias.gross@cs.fau.de
Friedrich-Alexander-University Erlangen-Nuremberg
Department of Computer Science

Tilo Müller
tilo.mueller@cs.fau.de
Friedrich-Alexander-University Erlangen-Nuremberg
Department of Computer Science

ABSTRACT

Apps written in JavaScript are an easy target for reverse engineering attacks, e.g. to steal the intellectual property or to create a clone of an app. Unprotected JavaScript apps even contain high level information such as developer comments, if those were not explicitly stripped. This fact becomes more and more important with the increasing popularity of JavaScript as language of choice for both web development and hybrid mobile apps. In this paper, we present a novel JavaScript obfuscator based on the Google Closure Compiler, which transforms readable JavaScript source code into a representation much harder to analyze for adversaries. We evaluate this obfuscator regarding its performance impact and its semantics-preserving property.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; Software reverse engineering;

KEYWORDS

Obfuscation, JavaScript, Hybrid Apps, Google Closure Compiler.

ACM Reference format:
Tobias Groß and Tilo Müller. 2017. Protecting JavaScript Apps from Code Analysis. In *Proceedings of SHCIS'17, Neuchâtel, Switzerland, June 21-22, 2017,* 6 pages.
DOI: 10.1145/3099012.3099018

1 INTRODUCTION

JavaScript has become an important programming language in the development of modern web applications as well as for hybrid mobile apps. Today, JavaScript is not only used to create modern websites in conjunction with HTML and CSS, but becomes increasingly popular for mobile apps based on hybrid frameworks such as Apache Cordova and Adobe PhoneGap. Those frameworks employ JavaScript as a language for cross-platform development that enable customers to develop a single code base for both Android and iOS by providing access to the native interface of the underlying OS through JavaScript.

SHCIS'17, Neuchâtel, Switzerland

978-1-4503-5271-0/17/06…$15.00
DOI: 10.1145/3099012.3099018

1.1 Motivation

The disadvantage implied when using JavaScript for cross-platform development is that JS files are necessarily delivered as plain source code, thus revealing all secrets of an app to its adversaries. When providing apps based on JavaScript, there is no compilation step involved as for apps based on Java, Objective-C or Swift, being transformed to either machine code (Objective-C and Swift) or bytecode (Java). Both, machine code and bytecode, are generally harder to reverse engineer than source code (although bytecode is known to be an easy target for reverse engineering, too).

Compilation steps always remove abstraction layers provided by the high-level language, e.g., type information and variable or function names, as well as control structures like if-statements and loops. From this point of view, compilation provides a basic obfuscation step, leaving JavaScript apps unprotected.

Using frameworks such as Cordova and PhoneGap, without protecting JavaScript code manually or with extra tools, even identifier names and comments are existent in the Android and iOS app packages. To protect the intellectual property of an app, and to harden it against repackaging attacks, it must be protected with an extra JavaScript obfuscator.

1.2 Contributions

In this paper, we present a JavaScript obfuscator combining strong obfuscation techniques previously mentioned in literature for other programming languages. Those techniques include (1) *identifier renaming*, (2) *string encryption*, (3) *function merging*, (4) *function inlining*, (5) *control flow flattening*, (6) *literal obfuscation*, (7) *anti-debugging* and (8) *packer methods*.

We evaluate the performance impact of the obfuscation techniques and, based on our results, propose different obfuscation levels that provide a good trade-off between security and performance for certain software parts such as crypto and UI.

From a technical point of view, our obfuscator is based on the Google Closure Compiler, thus being expandable with obfuscation plugins in the future.

2 ATTACKER MODEL

The scenario in which we want to protect apps from reverse engineering is the *malicious host scenario*. In this scenario, apps are provided as web content, presented by a browser or a WebView component controlled by an adversary. In practice, end consumers cannot be distinguished from adversaries when providing a web service or distributing apps via Google's and Apple's app stores.

Attackers do not only have arbitrary access to the entire app and can locate, copy and modify it, but also control the underlying host including its browsers, WebView components, libraries and even the OS kernel. Apparently, not all of those attacks can be

defeated from an unprivileged level such as a JavaScript sandbox. The time, however, that an attacker must invest to learn about a program's behavior and break its safety measures, can be increased considerably.

For JavaScript, most practical attacks are often performed by simplifying and beautifying (e.g. code formatting) the source code and looking at it afterwards (static analysis). Another popular approach of attackers is to utilize a JavaScript debugger (provided by either the browser or Node.js) to step through a running JavaScript app (dynamic analysis). Common attacks like those are in the main focus of our protection measures.

3 JAVASCRIPT OBFUSCATOR

The obfuscator tool is based on Google's Closure Compiler. The Google Closure Compiler provides source-to-source compilation for minification and optimization, meaning that it takes JS files as input and outputs transformed JS files that provide the same behavior. In general, the output code is smaller in size and will be executed faster by JavaScript interpreters [2].

Until today, however, the Google Closure Compiler does not provide transformations that primarily obfuscate code. To close this gap, we implemented eight obfuscation steps as compiler passes, which can be thought of as plugins. Those compiler passes are presented in the following subsections.

Figure 1 shows the order in which the compiler passes are applied to the input code to generate obfuscated code. Each transformation pass can be easily enabled by using the corresponding switch of the provided command line interface.

3.1 Identifier Renaming

This compiler pass renames all function, variable and parameter identifiers in the source code. Random generated or user defined names can be used as new names. For each identifier type the pool, which serves the new names can be set individually. This transformation relies on a namespace analysis, which is performed in front of the renaming, to rename each identifier according to its declaration.

By default, names which are extracted from the JavaScript framework AngularJS are used. But the implementation can use custom names by implementing classes of the `NameProvider` interface. Often, names generated by other JavaScript optimizers or obfuscators are short and meaningless. The usage of apparently meaningful names (extracted out of JavaScript libraries for example) can be an advantage. An adversary can be mislead, if the names of functions suggest other behavior then they actual have.

Another usable `NameProvider` generates names with a prefix `0x` (letter o x, not zero x) and a random integer written in base 16. These names are valid JavaScript identifier but an adversary will confuse these identifiers with hexadecimal number literals.

Namespace analysis create a namespace tree which match identifier to their declaration. From the root scope, each namespace is forced to rename their identifier declarations. Names which collide with existing valid names in the current namespace are discarded.

3.2 String Encryption

The string encryption transformation collects all string literals from the input code and encrypts them.

Before string collection, an abstract syntax tree (AST) visitor converts each `object.property` construct to an equivalent `object["property"]`. That allows to obfuscate the object attribute access operations with string encryption, too. Simultaneously to this transformation, the visitor collects all strings.

After string collection, all strings are concatenated to a central string. To separate the strings a suitable delimiter is determined, which is not part of any collected string. At start the delimiter is set to `"::"` and altered by inserting `"|"` as long as the delimiter is no more present in a collected string.

For encryption a random string `k` with the same character count as the concatenated string `m` is created. After that, a xor operation is performed, with `k` and `m` resulting in the encrypted string `c`.

Next, strings `k`, `c` and a decryption function are placed in the source script. The decryption function is implemented as template in JavaScript with placeholders for `k` and `c`. These placeholders are replaced with the actual values.

All used built-in functions in the decryption function, are accessed with strings resulting from concatenated characters. In conjunction with transformation identifier renaming, illegible statements are created.The decryption method is added to the input script and provides the decrypted strings at runtime.

As last step all collected string literals of the original code, are replaced with references to elements of the string array, created at runtime.

3.3 Function Merging

Function merging assembles suitable functions as new combined functions. This transformation is performed in three steps: bundle suitable functions, sort out unsatisfying functions and create merged functions from bundles.

To bundle functions, the scope tree created by namespace analysis is used. Functions defined in the same namespace are added to the same bundle. But only functions, which have no function declaration inside themselves are bundled. Since JavaScript has no static types, mostly every function can be merged.

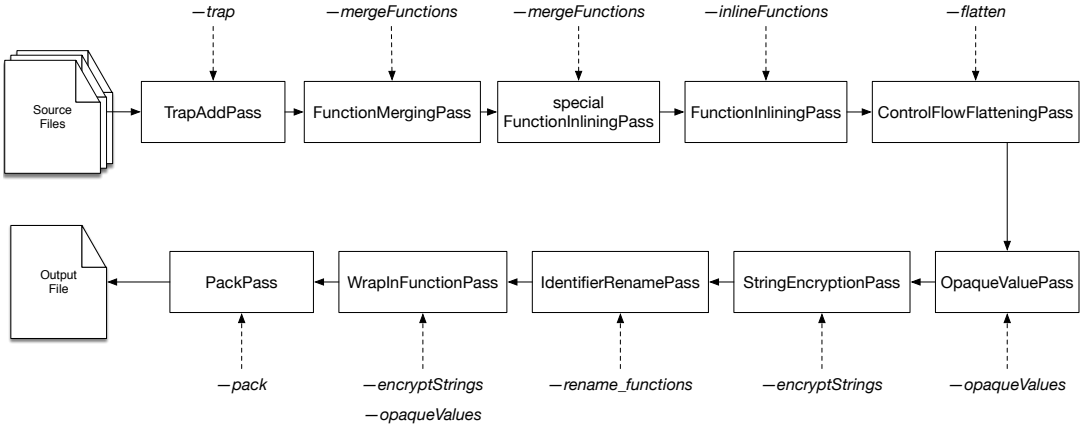
Functions, which use the `arguments` identifier are dropped, because this implementation can not handle merging functions with variable argument count. After that filtering of suitable functions, all bundles with less than two functions are dropped.

For each bundle, a new function is created. First, the parameter count of the function with most parameters is determined. An additional parameter `which` is added, that decides which merged function is executed. The body of the new function contains a function call of each merged function which is executed in condition to `which`. Each call is combined with a `return` which redirects the return value to the caller of the merged function.

After the creation of a merged function, all calls to the original functions are modified. Their references are altered to the new merged function with an additional constant, to control the executed code inside the merged function.

In conjunction with function inlining transformation, a real merging of the statements of each merged function is performed.

Figure 1: Order of obfuscation passes applied to source files.



3.4 Function Inlining

This code transformation was originally used as a compiler optimization technique. Later it was mentioned in the context of code obfuscation. [4, 5]

Function inlining replaces a call with the statements of the called function, among other small changes. The transformation is performed in three steps: find possible spots for inlining, prepare function to inline and carry out inlining.

Pairs of functions and calls, which are suitable for an inline transformation, are created. Only functions with no other function declarations inside their body are considered. Recursive functions are excluded for inlining, too.

The created pairs are checked. First all used identifiers in a function candidate which did not reference local definitions are collected. If one of these can not reference the right declaration at the destination of inlining, the pair is dropped.

Calls inside the head of a loops, for instance call `bar` in `while(bar()){...}`, are dropped, because inlining can not be performed semantics preserving there.

Expressions that evaluate to boolean values, including calls can not be inlined, too. There is short evaluation semantics which will only execute the call of function one in an expression like `one() || two()`. Since we have to place the function's statements before the expression, the semantics will be altered.

During the inlining process the return functionality of functions has to be replaced with a semantic equivalent construct. All returns are replaced by a labeled break and the body of the function is wrapped to a labeled block.

The function stopping semantic of a return inside the inlined code is emulated with a labeled break statement. Therefore, all statements of the function's body are copied into a labeled block. Every return statement is replaced by a labeled break related to the block. In addition to the exchange of return statements, a assignment to a variable providing the return value is put in front.

If a function uses the `arguments` identifier to obtain the call arguments, then it is replaced with semantic equivalent construct. After inlining the original function does not exist and an `arguments` variable must be provided explicitly. Semantic right behavior is ensured by replacing all *arguments* references with references to the new variable containing the arguments.

The last modification on a function candidate body is the renaming of local identifiers. Every local declared identifier its name collide in the new namespace, a new name is assigned.

As last step during this transformation, the modified function bodies are inserted at each possible spot for inlining. Parameters are exchanged with equally named variables. Such variables are assigned with the right values from the call arguments before the function statements starts.

The function's statements are appended as whole before the call statement. At last the call is replaced with an identifier which references the return value of the inlined function, if it is used in the context.

3.5 Control Flow Obfuscation

The implemented method of control flow obfuscation was mentioned by Wang et al. [9]. The resulting code consists of a continuous while loop and a switch construct inside the loop. A generated control flow graph (CFG) representation of the AST serves as input for the control flow transformation. A CFG consists of basic blocks with code that is executed sequential and edges which define the control flow between the basic blocks. These edges represent conditional and unconditional jumps.

This code transformation will turn every basic block into a case of a switch statement. At the end of each case, a variable is set which determines the next case (corresponding basic block) to execute. The switch statement is surrounded by a while loop which continuous executes the switch. Only return statements, are used to break the execution of switch and return from an function call.

The transformation process can be divided into three steps: preparations, create control flow graph and create switch table.

The preparation step transforms parts in the AST to semantic equivalent forms, so that the subsequent control flow graph generation has to cover less special cases. All for-in statements are converted to ordinary for loops, which allows to advect for-in

constructs in the switch table, too. That does not completely preserves the semantics of the original for-in loop [6]. If that destroys application behavior it has to be disabled.

As second preparation, all functions and try constructs are collected. A return statement is appended to each function with no return statement as last statement. This return is implicit existing in JavaScript semantics, anyway, but we need an explicit return to leave the enclosing endless loop of the switch construct.

As second step, for every function and every try construct, the control flow graph is generated separately.

As last step of this transformation, each basic block of a CFG is transformed to a case in the switch statement. At the end of each case a variable is set to identify the next case to execute. On a unconditional branch dead code is created which is never executed. This dead code sets `nextCase` to fool an analyst with additional control flow which does not really exist. To obfuscate the dead code, literal obfuscation should be applied afterwards, to create opaque misleading constructs for adversaries.

A conditional branch is constructed analogous. Instead of a `true` condition in the if statement, the condition of the jump is used and in both cases the `nextCase` variable is set properly.

In contrast to the switch table created for a function's CFG, the switch table for try construct's CFG can not be exited with a `return`. A `return` would exit the enclosing function of the try statement instead of only the statement itself. To break the execution of a switch table created from a try construct, a introduced variable control whether to switch is looped or not.

An example result produced by the control flow obfuscation transformation is displayed in Listing 1. There we see, that case numbers are assigned randomly and that harder to read code is created.

Listing 1: Control flow obfuscation example result.

```
function abc(par) {
  var ___pc = 985;
  while (true) {
    switch(___pc) {
      case 313:
        alert("function body");
        return;
      case 985:
        if (par) {
          ___pc = 588;
        } else {
          ___pc = 847;
        }
        break;
      case 588:
        alert(par);
        if (false) {
          ___pc = 588;
        } else {
          ___pc = 313;
        }
        break;
      case 847:
        alert("No parameter");
        if (false) {
          ___pc = 985;
        } else {
          ___pc = 313;
        }
        break;
    }
  }
}
```

3.6 Literal Obfuscation

This method was mentioned by Collberg and Nagra [3] as algorithm "OBF**WHKD**$_{\text{opaque}}$: Opaque Values from Array Aliasing". Especially in conjunction with control flow flattening this transformation provides a higher level of obfuscation to the source, because we can construct predicates out of the alias array. These opaque predicates obfuscate the control flow further more.

To obtain opaque values/predicates with this algorithm we first need to construct an array of integers. Such array, used to create opaque values, consist of different element types. There are elements which are equal to n mod m, called invariant. Ten different invariants are randomly defined for an alias array used in this implementation. Next we have the n , m elements with a constant value. The last type of elements are bogus elements, which are randomly assigned and should only confuse an adversary.

With such a constructed array we can construct predicates. For each element in the array the type is known, since they are defined at obfuscation time. If we want to construct an opaque true value we can write `a[0] % a[4] == a[16]` if `a[0]` is defined to be an invariant and `a[4]`, `a[16]` are the corresponding n and m values.

To create opaque values we can use the invariant properties of the array again. For example, 9 can be expressed as $2 * 2 * 2 + 1$ if we have defined invariants with n values of 1 and 2. Then $2 * 2 * 2 + 1$ can be expressed with array elements of the invariants: e.g. `a[1]%a[12]*a[2]%a[12]*a[7]%a[12]+a[0]%a[4]`.

To obfuscate the computations on the array elements most of the elements are exchanged during the program execution. We can exchange every element with a random integer as long as the defined invariants hold. Constant values are exchanged to, but it should only appear as exchange since we need them constant for calculations. In truth the same value is assigned, but calculated with array elements. Bogus elements are exchanged randomly.

To obfuscate the invariants of the array even further, we can add bogus assignments to array elements in parts of the program which are never executed. Such assignments will break the invariants but that did not bother, because they are never executed. The assigned bogus values are constructed with the help of array elements, too. It would be flashy if the bogus values are assigned direct (with a constant literal for example).

3.7 Anti Debugger Method

This transformation inserts code that will hamper code analysis with standard JavaScript debuggers provided by Chrome, Firefox or other JavaScript interpreters. The inserted code will cause a debugger to break on unappealing locations plenty of times. The breaks are triggered with the `debugger` keyword.

Listing 2 shows the function declaration of the debugger trap function. On line 8 we can see the core of the debugger trap. It will construct an anonymous function with `debugger` statement as function body and calls the created function. With this odd method of function declaration, a debugger will leave less trace where the code is located.

Listing 2: Debugger trap code.

```
1  (function trap(){
2      try {
3          (function recTrap() {
4              (function(){}).constructor('debugger')();
5              recTrap();
6          }());
7      } catch(e) {
8          setTimeout(trap, 3000);
9      }
10 })();
```

This functionality is wrapped in a recursive function `recTrap` which will call itself till a stack overflow occurs. A resulting exception is cached and sets a timeout which will call the trap function after three seconds again (see line 8). The enormous growing callstack hampers an analyst additionally. Next application code is execution is continued before, after three seconds the debugger trap code is executed again.

3.8 Packer

With this transformation the application code is entirely encrypted. The decryption of the code is performed at runtime. Thereby a key is generated and xored with the encrypted application code. The generated key is dependent on the decryption function definition and checks of the used build-in functions.

Functions/methods `eval`, `toString`, `substring`, `fromCharCode` and `charCodeAt` are tested for referencing the original function. The tests utilize that build-in functions have no `prototype` property. These checks are important because an adversary can easily exploit them to circumvent the obfuscation added with the packer transformation and receive the decrypted application code. If tests are not passed, the key is generated false.

They generated key is additionally dependent on the decryption function's definition. The `toString` method is used on the function identifier to receive the function definition as string. This proceeding can detect modifications of the decryption method and we obtain a simple tamper proof mechanism. In addition, the decryption method contains the anti debugger method mentioned before.

If the right key is generated (no tampering detected and valid build-in functions), the application code is decrypted and then executed with `eval`. The entire decryption function is obfuscated with all transformations mentioned before to veil the functionality and hamper static analysis.

4 EVALUATION

The evaluation is performed on obfuscated code with five reasonable transformation combinations: low (Identifier Renaming, String Encryption), mediumA (low + Function Merging, Function Inlining), mediumB (low + Control Flow Obfuscation, Literal Obfuscation), high(mediumB + Function Merging) and max (all transformations). The configurations labels refer to the amount of achieved obfuscation, if using the option.

Performance impact appear in a higher (sometimes lower) execution time of an obfuscated program instead of the original one. If we measure the execution time Δt_{obf} of the obfuscated code and the execution time Δt_{src} of the original code, than the performance impact I is determined as $I = \frac{\Delta t_{obf}}{\Delta t_{src}}$.

I	low	mediumA	mediumB	high	max
Safari	1.1	1.1	4.5	1303.8	1299.9
Firefox	1.5	1.7	9.0	1017.2	1017.5
Chrome	1.8	2.0	17.5	17383.9	16878,9

Table 1: Performance impact on bullet chart.

To determine the range of performance impacts for several implemented obfuscation transformations, D3.js and CryptoJS library is instrumented.

To measure the impact on D3.js, five different diagrams are drawn: Bubble Chart, Bullet Charts, Chord Diagram, Circle Packing and Node-Link Tree.[1]

For each diagram, the draw time is measured. Starting at the point where the script is loaded and ending at the point the complete diagram is shown on the screen. The measuring is performed in three JavaScript runtimes (particularly browsers) for each diagram: Safari, Chrome and Firefox The performance impact on D3.js is determined for five obfuscation configuration on five diagrams on three runtimes. This leads to 45 measurements of Δt_{obf}.

The used obfuscator configurations are suggestive combinations of individual obfuscation methods. The groups are combinations of code transformations, which complement each other well and range from low obfuscation to high obfuscation.

The measurements revealed highly scattered execution times. One diagram's measured execution times exhibits a quartile distance $\Delta t_{0,75} - \Delta t_{0,25}$ in the region of 10^6. That shows that the execution time is not exact determinable and gives only rough estimate of the performance impact. Execution time measurements on different diagrams confirmed that vague behavior: The performance impact is not exactly determinable for a specific obfuscations.

The determined median values of each measurement series are used to calculate the performance impact I. Performance impacts of each configuration on the drawn Bullet Chart is stated in Table 1. There we can see, that every runtime handles the obfuscated code different. Firefox and Chrome handle the obfuscations low and mediumA alike (Chrome is a little bit slower), but high and max has a ten times greater performance loss on Chrome in contrast to Safari/Firefox. In other cases the relation changes dependent on the obfuscation configuration: Where Safari is faster than Firefox in configurations low, mediumA and mediumB, Firefox is faster in the configurations high and max. If each runtime is considered separately, you can see that the degree of obfuscation is dependent on the degree of performance loss. For most cases noticeable, the higher the obfuscation, the higher the performance loss.

Over all diagrams we had experienced that the obfuscations low and mediumA have a fair amount of performance loss. From option mediumB to max the performance loss is vast.

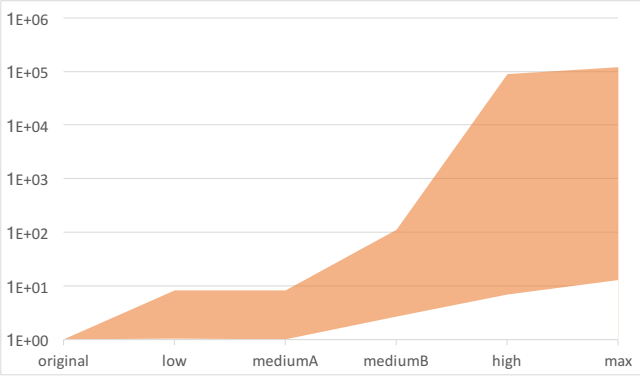
To obtain a complete range of performance impacts of each obfuscation configuration different code styles have to be investigated. We assume that different input code (in the meaning of style) will react with a different amount of performance impact. For example,

[1] https://github.com/d3/d3/wiki/Gallery

I	low	mediumA	mediumB	high	max
SHA256	1.3	1.2	11.3	10.1	13.0
SHA3	8.3	8.3	110.5	114.6	110.8
AES	1.1	1.1	6.7	6.9	814.1
DES	1.0	1.0	7.5	7.5	286.0

Table 2: CryptoJS performance impact.

Figure 2: Range of performance impact of several obfuscation configurations.



an application with complex control flow will lead to much more performance loss by control flow flattening instead of code with simple control flow. Therefore heterogeneous code is needed as benchmark of the performance impact.

To complement the measures on D3.js diagrams, measures on CryptoJS are performed. Cryptographic code should be different to user interface drawing code in the meaning of code-style (more calculations in the cryptographic code, more function calls in the drawing code). Four functions are chosen are measured: SHA256, SHA3 512 Bit, AES and DES. The execution time is measured by running the cryptographic functions in Firefox.

With the determined median values as Δt the performance impact is calculated for each cryptographic function and obfuscation configuration. The result is stated in Table 2. This reveals again, that different code experiences different performance impact. For example SHA256 implementation experience a 60 times lower performance loss than AES implementation.

To complete the consideration of performance impact Figure 2 displays the complete range of experienced performance impacts. This elucidates the dynamic performance cost of the obfuscation configurations. As assumed, some code is more resistant against the negative influence of obfuscation transformation than other code. For example the minimum performance loss of the max configuration is about the same as the maximum performance loss of the mediumA or low configuration. On the other side there is code that experiences a 10.000 times higher performance loss with the same obfuscator configuration (max). The second factor which influences the performance impact is the target runtime of the obfuscated JavaScript code. The runtime influence the execution time of obfuscated code enormous.

5 RELATED WORK

Bertholon et al. [1] used evolutionary algorithms together with obfuscation methods like identifier mangling, statement outlining, bogus code insertion and string obfuscation, to implement a JavaScript obfuscator. This approach allows to define the degree of obfuscation with software metrics and a maximum performance loss which is not allowed to be exceeded. In contrast to this paper, they focused on the evolutionary algorithms and implemented partly other obfuscation transformations.

Kolisar [7] took a steganographic approach to obfuscate JavaScript code. Therefore data is encoded with whitespaces within HTML web pages. In contrast to this paper, the focus lies on a specific method of string encryption.

Schrittwieser et al. [8] considered different scientific works on obfuscation and software analysis methods to give an estimate on obfuscation methods' strength. In contrast to this paper, the treated obfuscation methods are not focused on a specific programming language / machine code. In conclusion, there exist obfuscation methods which protect against specific analysis methods.

6 CONCLUSION

With this paper, we showed which steps can be performed to defeat analysis of JavaScript code. Eight different obfuscation methods and an optimal order in which they can be applied were mentioned. Together, the methods compliment each other well, in the aspect of obfuscation.

We showed that is possible to adapt these eight transformations, mentioned in the literature for JavaScript code. Additionally, the implemented methods are stated tailored to JavaScript.

The evaluation yield that the performance loss can be enormous if all implemented transformations are applied. At obfuscating JavaScript code, applying all transformations should be restricted to important code locations, which do not react critical to performance loss. There are, however, obfuscations which can be applied to the whole application code, since they produce only little performance loss.

REFERENCES

[1] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. 2013. Jshadobf: A javascript obfuscator based on multi-objective optimization algorithms. In *International Conference on Network and System Security*. Springer, 336–349.
[2] Michael Bolin. 2010. *Closure: the definitive guide* (1. ed ed.). O'Reilly, Sebastopol, Calif.
[3] Christian Collberg and Jasvir Nagra. 2010. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection.* Addison-Wesley, Upper Saddle River, NJ.
[4] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations.* Technical Report. Department of Computer Science, The University of Auckland, New Zealand.
[5] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Breaking abstractions and unstructuring data structures. In *Computer Languages, 1998. Proceedings. 1998 International Conference on.* IEEE, 28–38.
[6] David Flanagan. 2011. *JavaScript: the definitive guide* (6th ed ed.). O'Reilly, Beijing ; Sebastopol, CA.
[7] Kolisar. 2008. WhiteSpace: A Different Approach to JavaScript Obfuscation. (2008). DEFCON 16.
[8] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2015. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *Comput. Surveys* (2015), 1–40.
[9] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software tamper resistance: Obstructing static analysis of programs.* Technical Report. Technical Report CS-2000-12, University of Virginia, 12 2000.