Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly

Alan Romano
University at Buffalo, SUNY
Buffalo, United States
alanroma@buffalo.edu

Daniel Lehmann
University of Stuttgart
Stuttgart, Germany
mail@dlehmann.eu

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

Weihang Wang
University at Buffalo, SUNY
Buffalo, United States
weihangw@buffalo.edu

***Abstract*—To protect web users from malicious JavaScript code, various malware detectors have been proposed, which analyze and classify code as malicious or benign. State-of-the-art detectors focus on JavaScript as the only target language. However, WebAssembly provides attackers a new and so far unexplored opportunity for evading malware detectors. This paper presents Wobfuscator, the first technique for evading static JavaScript malware detection by moving parts of the computation into WebAssembly. The core of the technique is a set of code transformations that translate carefully selected parts of behavior implemented in JavaScript into WebAssembly. The approach is opportunistic in the sense that it uses WebAssembly where it helps to evade malware detection without compromising the correctness of the code. Evaluating our approach with a dataset of 43,499 malicious and 149,677 benign JavaScript files, as well as six popular JavaScript libraries reveals that our approach is effective at evading state-of-the-art, learning-based static malware detectors; the obfuscation is semantic-preserving; and our approach has small overhead, making it practical for use in real-world programs. By pinpointing limitations of current malware detectors, our work motivates future efforts on detecting multi-language malware in the web.**

I. Introduction

The omnipresence of the web makes client-side web applications an attractive target for attackers. As a result, various kinds of attacks target the browser, e.g., drive-by malware [48], [23], [43], malicious code deployed via script-based browser augmentation markets [58], browser-based cryptomining without user consent [40], [52], [34], malicious browser extensions [21], [65], and browser-based phishing [14]. A recent report estimates that orchestrated phishing campaigns alone create 1.7 to 2 million malicious payload URLs each month [44]. Beyond such obviously malicious activities, many websites employ techniques that are unwanted by users, such as extensive browser fingerprinting [30] or tracking [25].

To protect users against executing malicious scripts in their browser, JavaScript malware detectors warn about such scripts before or while executing them. One line of work statically analyzes scripts before they are executed [24], [50], [56], [28], [27], e.g., by intercepting them already in the network, as part of a browser, or as part of a separate anti-virus tool. Another line of work dynamically analyzes scripts [36], e.g., by instrumenting the code or via a browser extension. To reduce the runtime overhead imposed by dynamic analysis-based malware detectors, static detectors often serve as a first line of defense, e.g., by dynamically analyzing only those scripts that are deemed dangerous by a static analysis.

To effectively attack users despite the presence of malware detectors, attackers try to hide the maliciousness of scripts via obfuscation and evasion techniques [53], [55]. Progress by attackers and defenders leads to an arms race between increasingly sophisticated obfuscation and evasion techniques on one hand and increasingly effective malware detectors on the other hand. Currently, the most effective malware detection techniques use learning-based classifiers to distinguish malicious from benign scripts [24], [50], [28], [27]. These approaches extract a set of features from a given JavaScript file, e.g., n-grams of code tokens or AST-based features, or feed the JavaScript code into a deep neural network [63] to determine whether the file is likely to be malicious.

While the focus on JavaScript historically makes sense, JavaScript is not the only language of the client-side web anymore. WebAssembly [31] is another language that is widely available in browsers. First announced in 2015, WebAssembly is supported by all major browsers since November 2017, and available in 94% of all global browser installations as of August 2021[1]. The language provides an efficient compilation target for computation-intensive libraries written in languages such as C and C++. In addition to the many positive uses of WebAssembly, the language provides a new opportunity to attackers for evading malware detectors – an opportunity that, to the best of our knowledge, has not yet been explored.

This paper presents the first technique for evading JavaScript malware detection by moving parts of the computation into WebAssembly. We describe Wobfuscator, a code obfuscation technique that transforms a given JavaScript file into a new JavaScript file and a set of WebAssembly modules. By changing the malicious JavaScript code, our work aims at evading static malware detectors. The rationale is that static detectors may be used on their own or serve as a filter for which scripts to analyze dynamically. That is, evading static malware detectors gives a huge benefit to attackers.

Transforming parts of a JavaScript file into WebAssembly is far from trivial. JavaScript is dynamically typed, has complex objects, and provides direct access to browser APIs. In contrast, WebAssembly is statically typed, has only four low-

[1] https://caniuse.com/?search=WebAssembly

level, primitive data types, and it can access browser APIs only indirectly by importing them from JavaScript. Because of these fundamental differences, general JavaScript-to-WebAssembly translation is practically impossible, which is reflected in the fact that WebAssembly never has been touted as a replacement for JavaScript but as a way to complement it [31].

The core technical contribution of this paper is a set of code transformations that extract carefully selected parts of behavior implemented in JavaScript for translation into WebAssembly. The approach is opportunistic in the sense that it translates JavaScript to WebAssembly where it helps to evade malware detectors, without compromising the correctness of the code. For example, we present a transformation that extracts function calls into a WebAssembly module, obfuscating if and when a script calls a particular function. Other transformations aim at obfuscating string literals, control-flow statements, and array initializations. Preconditions guard each transformation to ensure the original behavior is preserved, a property we consider crucial for an attacker to use an obfuscation technique.

Our work relates to existing obfuscation techniques for JavaScript. For example, Fass et al. fuse malicious code into benign code while preserving the AST of the benign code [26]. For a more comprehensive overview of obfuscation techniques and their prevalence, we refer readers to other work [66], [55]. All of these approaches obfuscate code via transformations *within* the JavaScript language, whereas Wobfuscator exploits the availability of WebAssembly to obfuscate code by translating *beyond* JavaScript. A paper by Wang et al. shares the general idea of obfuscating code written in one language by translating parts of it into another language [60]. However, they describe partially translating C code into Prolog, which does not address the unique challenges of obfuscating JavaScript via partial translation to WebAssembly.

We evaluate Wobfuscator with a dataset of 43,499 malicious and 149,677 benign JavaScript files, as well as six popular JavaScript libraries. Our results show the following. First, the approach is effective at evading state-of-the-art, learning-based static malware detectors: Applying our transformations reduces the recall of the four studied detectors [50], [24], [28], [27] to 0.18, 0.63, 0.18, and 0.00, respectively. Second, the obfuscation preserves the semantics of the transformed code: Obfuscating six popular JavaScript libraries and running their 2,017 tests shows no observable changes in the behavior of the tested code. Finally, we find that our tool only takes on average 8.9 seconds to apply all the transformations to a project (with on average 4,152 lines of code) and adds on average 31.07% of overhead during runtime. Overall, these results show that Wobfuscator is practical for use in real-world programs.

In summary, this paper contributes the following:

- The first technique to use WebAssembly as a means for obfuscating the behavior of malicious JavaScript code.
- A set of code transformations that translates carefully selected JavaScript code locations into WebAssembly.
- A comprehensive evaluation showing that the approach effectively evades state-of-the-art static malware detectors while preserving the semantics of the original code.

Our experiment results are publicly available: https://github.com/js2wasm-obfuscator/translator

II. Background on WebAssembly

We present a brief introduction of the core concepts and syntax of WebAssembly. For space reasons, we refer the reader to the original publication [31], official website [10], or specification [64] for more elaborate explanations.

WebAssembly is a low-level byte code language. WebAssembly programs are distributed as binaries that are compact to send over the network and quick to parse.

Each WebAssembly program is a single-file *module*, organized into several *sections*. Most importantly, the *code* section contains all functions with their bodies. The *global* section contains scalar global variables. The *memory* section declares a linear, byte-addressable memory, of which parts can be initialized with the *data* section.

WebAssembly instructions operate at a low level of abstraction, without, e.g., classes or complex objects. Instructions and functions are statically typed, but there are only four *primitive types*: 32- and 64-bit integer and floating-point values, respectively. Instructions are executed on a virtual *stack machine*, i.e., they pop their operands and push their results onto an implicit evaluation stack. Instructions are simple and designed to map closely to hardware instructions, e.g., a WebAssembly `i32.add` instruction would be translated into an x86 `addl`. Since there is no garbage collector and only primitive scalar types, complex objects (strings, arrays, records, etc.) are stored into program-organized *linear memory*, which is essentially a growable array of untyped bytes.

WebAssembly does not have a standard library, and for interaction with the "outside world", WebAssembly modules need to import functions from the *host environment*. In the browser, that host environment is JavaScript, so any JavaScript function can be called from WebAssembly. Non-primitive data needs to be marshalled through the WebAssembly memory however (which can be accessed from JavaScript as well). The WebAssembly API in JavaScript provides functions to compile and instantiate (that is, fulfill the imports of) WebAssembly modules and call exported WebAssembly functions.

III. Threat Model

Our work is about malicious or unwanted code delivered as part of a client-side web application. A website may deliver such code intentionally, e.g., because the domain is controlled by the attacker, or unintentionally, e.g., via third-party scripts or advertisements. We assume that the attacker controls the malicious code on the server side, and hence, can freely modify it. In particular, they can transform the source code, split one file into multiple files, or merge multiple code files.

On the defense side, we assume a static malware detector that scans the client-side code of a web application before it executes in the browser. The mechanism the malware detector uses to intercept and check the code is irrelevant to us. For example, it may be implemented as a network proxy that scans

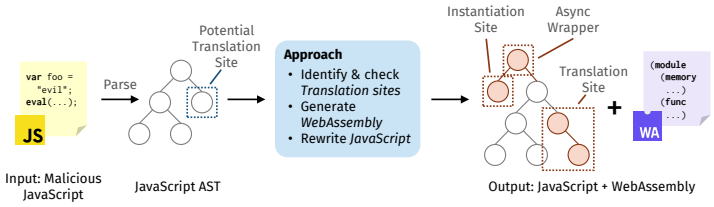

Fig. 1. Overview of Wobfuscator.

files before they even reach the client's machine, as an anti-virus tool on the client machine, or as a browser extension. Malware detection or mitigation techniques that go beyond static analysis, e.g., based on analyzing the executing of client-side code, are beyond the scope of this work.

The goal of the attacker is to bypass the malware detector. To this end, the attacker may assume that the targeted browser supports WebAssembly, which is the case for almost all of today's browsers. The attacker does not need to have access to the malware detector or, in the case a machine learning-based detector, to the data it is trained on.

IV. Approach

A. Overview and Challenges

Figure 1 gives an overview of Wobfuscator. The input is a JavaScript file, which we parse into an AST. Next, the approach identifies potential *translation sites*, i.e., code locations that (i) are relevant for detecting malicious code and (ii) can be translated into WebAssembly in a semantics-preserving way. Instead of aiming at a general JavaScript-to-WebAssembly translation, the approach opportunistically targets only those code locations that fulfill these two requirements.

To move behavior into WebAssembly, Wobfuscator generates WebAssembly code for each translation site and then transforms the JavaScript AST to utilize the generated code. The AST is transformed in three ways. First, at an *instantiation site*, we add code to load the WebAssembly module into the application. Second, at each of the selected translation sites, we modify the code to access properties and functions provided by the WebAssembly module(s). Third, at the root node of the AST, we conditionally wrap the script into an anonymous, `async` function, referred to as the *async wrapper*, to support asynchronous keywords in the code. The remainder of this section explains these transformations in detail. Finally, the output of Wobfuscator is the transformed JavaScript code along with one or more generated WebAssembly modules.

We encountered several challenges in designing transformations that move JavaScript behavior into WebAssembly modules. One key challenge is the fact that completely translating general JavaScript code to WebAssembly is impossible due to the limited set of features in WebAssembly. One example is dynamically generated code, which is enabled in JavaScript through the infamous `eval` function, but not supported in WebAssembly. JavaScript supports rather complex rules, e.g., function scope for `var`-bound variables, hoisting of functions, and closures. WebAssembly has only three storage locations: global module variables, local variables for each function, and raw byte memory. In addition, WebAssembly only has limited control-flow instructions that do not fully replicate all available JavaScript constructs, such as `try/catch` statements, `for-of` loops, and `Promises`. Another challenge is the limited data types that can be passed between WebAssembly and JavaScript. In the initial version of WebAssembly, functions can only accept and return numeric data types, which cannot replicate the complex objects that JavaScript supports. Finally, different browsers can impose file-size limits on the WebAssembly modules used, so we must workaround this limitation for our technique to be general.

Because of these and other differences, Wobfuscator is based not on complete but *opportunistic translation*, i.e., transforming code where it helps to evade malware detectors, without sacrificing correctness.

B. Transformations

The goal of our approach is to generate WebAssembly modules that can reproduce the functional behavior of specific JavaScript code snippets. To produce these modules, Wobfuscator is constructed on a core set of transformation rules:

Definition 1 (Transformation rule)**.** A transformation rule is a tuple (L, t, p) where:

- L represents a set of code locations where the transformation may apply,
- t is a transformation function that maps JavaScript at a code location in L to rewritten JavaScript code and one or more WebAssembly modules, and
- p is a precondition for applying t, expressed as a predicate on a code location and its surrounding context.

We present seven transformation rules that target different language features of JavaScript. The transformation rules fall into three categories. First, we present rules to obfuscate data that gets defined and used by a JavaScript file. These rules target string literals (Section IV-B1) and array initialization code (Section IV-B2). Second, we present rules to obfuscate function calls. These rules hide suspicious function calls (Section IV-B3) or any function call (Section IV-B4). Finally, we present rules to obfuscate the control flow in the given code. These rules target if statements (Section IV-B5), for loops (Section IV-B6), and while loops (Section IV-B7). Table I illustrates the transformation rules.

The transformation rules use several JavaScript primitives to interact with WebAssembly:

- `instanWasm(source, impObj)` instantiates a WebAssembly module from `source` and returns the module. The optional parameter `impObj` is an object containing the functions to be imported into the created WebAssembly module. We use two variants of this primitive which instantiate the module synchronously and asynchronously (Section IV-C).
- `loadStrFromBuf(buffer, startIndex)` creates a string from the `buffer` starting from the byte offset `startIndex` and ending at the first null byte (i.e., `\00`)

TABLE I
TRANSFORMATION FUNCTIONS.

Rule	JavaScript (Before)	JavaScript (After)	WebAssembly (After)

T1-StringLiteral	`var s = "lit";`	`// Instantiation Site`
		`let m = instanWasm(source);`
		`let buf = m.instance.exports.memory.buffer;`
		`// Translation Site`
		`let startInd = m.instance.exports.d1;`
		`var s = loadStrFromBuf(buf, startInd);`
			`(global $d1 (export "d1") (mut i32) (i32.const 0))`
			`(memory (export "memory") 1)`
			`(data $data0 (i32.const 0) "lit\00")`

T2-ArrayInitialization	`var arr = new Array();`
	`arr[i1] = num1;`
	`arr[i2] = num2;`
		`// Instantiation Site`
		`let m = instanWasm(source);`
		`let buf = m.instance.exports.memory.buffer;`
		`// Translation Site`
		`m.instance.exports.f();`
		`var arr = loadArrFromBuf(buf, startInd, len);`
			`(memory (export "memory") 1)`
			`(func $f (export "f")`
			`  i32.const $i1`
			`  i32.const $num1`
			`  i32.store`
			`  i32.const $i2`
			`  i32.const $num2`
			`  i32.store)`

T3-FunctionName	`eval(str);`	`// Instantiation Site`
		`let m = instanWasm(source);`
		`let buf = m.instance.exports.memory.buffer;`
		`// Translation Site`
		`let startInd = m.instance.exports.d1;`
		`window[loadStrFromBuf(buf, startInd)](str);`
			`(global $d1 (export "d1") (mut i32) (i32.const 0))`
			`(memory (export "memory") 1)`
			`(data $data0 (i32.const 0) "eval\00"))`

T4-CallExpression(a)	`f(a);`	`// Translation Site`
		`let impObj = {imports: {impFunc: () => f(a)}};`
		`let m = instanWasm(source, impObj);`
		`m.instance.exports.f0();`
			`(func $f0 (export "f0")`
			`  call $impFunc) ;; JS import`

T4-CallExpression(b)	`let r = f(a);`	`// Translation Site`
		`let impObj = {imports: {impFunc: f}};`
		`let m = instanWasm(source, impObj);`
		`let r = m.instance.exports.f0(a);`
			`(func $f0 (export "f0") (param externref) (result externref)`
			`  local.get $p`
			`  call $impFunc) ;; JS import`

T5-IfStatement	`if(cond) {`
	`  stmt1; stmt2; ...`
	`} else {`
	`  stmt3; stmt4; ...`
	`}`
		`// Translation Site`
		`let impObj = {imports: {`
		`  imp1:() => {stmt1; stmt2; ...},`
		`  imp2:() => {stmt3; stmt4; ...}}};`
		`let m = instanWasm(source, impObj);`
		`m.instance.exports.f(cond ? 1 : 0);`
			`(func $f (export "f0") (param $p)`
			`  local.get $p`
			`  if ;; label = @1`
			`  call $imp1 ;; JS import`
			`  else`
			`  call $imp2 ;; JS import`
			`  end)`

T6-ForStatement	`for(init;cond;incre) {`
	`  stmt1; stmt2; ...`
	`}`
		`// Translation Site`
		`init;`
		`let impObj = {`
		`  imports: {`
		`  cond:() => {return cond ? 1 : 0},`
		`  incre:() => {incre},`
		`  body:() => {stmt1; stmt2; ...}`
		`  }`
		`};`
		`let m = instanWasm(source, impObj);`
		`m.instance.exports.f();`
			`(func $f (export "f0")`
			`  block $L0`
			`  loop $L1`
			`  call $cond ;; JS import`
			`  i32.eqz`
			`  br_if $L0`
			`  call $body ;; JS import`
			`  call $incre ;; JS import`
			`  br $L1`
			`  end`
			`  end)`

T7-WhileStatement	`while(cond) {`
	`  stmt1; stmt2; ...`
	`}`
		`// Translation Site`
		`let impObj = {`
		`  imports: {`
		`  cond:() => {return cond ? 1 : 0},`
		`  body:() => {stmt1; stmt2; ...}`
		`  }`
		`};`
		`let m = instanWasm(source, impObj);`
		`m.instance.exports.f();`
			`(func $f (export "f0") (param $p)`
			`  block $L0`
			`  loop $L1`
			`  call $cond ;; JS import`
			`  i32.eqz`
			`  br_if $L0`
			`  call $body ;; JS import`
			`  br $L1`
			`  end`
			`  end)`

after `startIndex`, where `buffer` is the WebAssembly module linear memory.

- `loadArrFromBuf(buffer, startIndex, length)` creates an array from `buffer` of size `length` starting from the byte offset `startIndex`, where `buffer` is the WebAssembly linear memory.

1) Obfuscating String Literals: JavaScript malware frequently uses encoded strings to hide malicious code [66]. These encoded strings can be critical for malware detectors which learn the string patterns and their encoding schemes [24], [54]. To evade the detection of encoded strings, we define a transformation rule *T1-StringLiteral* (L_{T1}, t_{T1}, p_{T1}) where:

L_{T1}. The code locations where transformation rule T1 may apply are all AST nodes of *Literal* type with string values.

t_{T1}. The transformation function is defined in row T1-StringLiteral in Table I. To obfuscate a string literal "`lit`", t_{T1} generates a WebAssembly module that defines a `memory` and exports it to JavaScript (line 2). The memory is used to store the string literal "`lit`" at offset `0` (line 3). To reconstruct this string in JavaScript, a variable `$d1` containing the offset is defined and exported (line 1). Each string is terminated with a null byte (i.e., `\00`), so it can be reconstructed by reading the linear memory from the starting index until the first null byte is found. If multiple strings in the input JavaScript program are to be transformed, they are all stored in a single WebAssembly module. In this case, multiple variables can be defined for each string stored (e.g., `$d1`, `$d2`, ...). The variable `buf` points to the

memory exported by WebAssembly (line 3). At the translation site, a variable `startInd` gets the starting index of the string literal stored in linear memory (line 5). Finally, the string "`lit`" is reconstructed using the primitive `loadStrFromBuf` with arguments `buf` and `startInd` (line 6).

p_{T1}. This transformation can be applied in locations where JavaScript allows for replacing a string literal with a function call. Specifically, this excludes: (i) string literals used in `import` or `require` statements, as such strings should be known when bundling modules together; (ii) string literals used as property names in object expressions as return values from function calls cannot be used as object keys.

2) Obfuscating Arrays: Malicious files often contain arrays of numeric literals representing character codes used to reconstruct malicious strings. To obfuscate these arrays, we exploit the fact that the linear memory of WebAssembly modules is implemented through an array buffer, which can naturally map to a JavaScript numeric array, in transformation rule *T2-ArrayInitialization* (L_{T2}, t_{T2}, p_{T2}) where:

L_{T2}. T2 may apply on *NewExpression* AST nodes (e.g., `new Array`) or *ArrayExpression* AST nodes representing array literal expressions (e.g., `[1,2,3]`). In addition, the transformation also condenses any following *AssignmentExpression* nodes used to initialize the array values into a single JavaScript function call (e.g., `arr[1] = 42; arr[2] = 97;...`).

t_{T2}. The transformation t_{T2} is defined in row T2-ArrayInitialization in Table I. The original JavaScript code creates an array `arr` and initializes the elements `arr[i1]` and `arr[i2]` with numerical values `num1` and `num2`, respectively. After the transformation, t_{T2} produces a WebAssembly module that creates a `memory` and exports it (line 1). A function `$f` is defined which stores the numbers, `num1` and `num2`, at the specified offsets, `i1` and `i2`, inside the linear memory (lines 2-8). The transformed JavaScript code instantiates a WebAssembly module (line 2) and creates a variable to read from the exported memory (line 3). Similar to T1-StringLiterals, if there are multiple array initializations to be transformed, only one WebAssembly module is created at the instantiation site. At the translation site, the export function `f` is called to write `num1` and `num2` to the linear memory at offset `i1` and `i2` (line 5). Finally, `loadArrFromBuf()` returns a JavaScript `Array` object containing the numerical values copied from the linear memory buffer, and this array is assigned to `arr` (line 6). This function requires the starting index of the array in linear memory and the length of the array. Both of these values are calculated by Wobfuscator and inserted into each call to `loadArrFromBuf()`. We ensure that a JavaScript `Array` object is returned as the original code using the array will need to access the standard properties and methods of an `Array`.

p_{T2}. This transformation can be applied to arrays initialized with numeric literals. Since WebAssembly only has numeric data types we only store numeric literals in memory using the operators `.const` and `.store`. Specifically, we apply the transformation only if the array initialization is one of the following: (i) a `new Array` expression followed by assignment statements inserting only numeric literals; (ii) a `new Array` expression with numeric literal arguments; (iii) an *ArrayExpression* only containing numeric literals.

3) Obfuscating Function Names: Several built-in JavaScript functions, such as the notorious `eval` function, are commonly exploited by attackers. As a result, detectors may consider the names of these built-in functions suspicious and use them as part of the signatures for malware detection [50], [24], [17]. To evade detection, we remove suspicious function names from the JavaScript code through a transformation rule *T3-FunctionName* (L_{T3}, p_{T3}, t_{T3}) where:

L_{T3}. T3 may apply on *CallExpression* nodes or *NewExpression* nodes that contain specific identifier names.

t_{T3}. The transformation function is defined in row T3-FunctionName in Table I. To obfuscate the function name `eval`, t_{T3} removes the function name from JavaScript and stores it in WebAssembly linear memory. In the WebAssembly code, a global variable `$d1` is defined and exported with a value of `0` (line 1), which is the starting index of the function name "`eval`" stored in the linear memory. Next, a `memory` is created and exported (line 2). To initialize the linear memory, a data section is defined that contains "`eval`" at offset `0` (line 3). In the transformed JavaScript code, it instantiates a WebAssembly module (line 2) and defines a variable to access the linear memory (line 3). The variable `startInd` is assigned the value of the exported `d1` that contains the starting index of "`eval`" in the linear memory (line 5). Finally, `loadStrFromBuf` is used to create the string "`eval`", and `eval` is called from the `window` object with `str`, the same argument used in the original `eval()` (line 6).

p_{T3}. This transformation can be applied to call expressions or new expressions referencing global functions accessible through the `window` object. Specifically, we identify eight global functions commonly used in malicious files and apply the transformation only if the identifier is in the following list: `eval`, `escape`, `atob`, `btoa`, `WScript`, `unescape`, `escape`, `Function`, and `ActiveXObject`. While these functions are not inherently malicious, many of the analyzed malware files use these functions to decode and execute hidden code.

4) Obfuscating Calls: Aside from the suspicious functions, we construct a transformation rule T4-CallExpression for general JavaScript function calls. This transformation converts function calls in JavaScript into a call of an exported WebAssembly function, which in turn performs a function call in WebAssembly to an imported JavaScript function. Unlike T3-FunctionName, which completely removes suspicious function names from JavaScript, this transformation modifies the context of the function being used in call sites that AST-based malware detectors use when scanning for malicious code.

There is a trade-off in this transformation between compatibility with the WebAssembly Minimum Viable Product (MVP) version and the amount of transformable function calls. Thus, we create two variations of this transformation: T4-CallExpression(a) is fully compatible with the WebAssembly MVP (i.e., uses no language extensions) but can only be applied on functions that do not return a value; T4-CallExpression(b) transforms functions with return values but

requires the WebAssembly Reference Types proposal [9]. Firefox and Chrome enable this proposal by default.

T4-CallExpression(a) is a transformation rule $(L_{T4a}, p_{T4a}, t_{T4a})$ where:

L_{T4a}. L_{T4a} are *CallExpression* nodes containing the identifier and arguments of a function call.

t_{T4a}. t_{T4a} is defined in row T4-CallExpression(a) in Table I. To obfuscate a JavaScript function call `f(a)`, t_{T4a} moves the function call into an anonymous function that is imported into WebAssembly (line 2). At the original call site, a WebAssembly export function `f0` is called (line 4). In the WebAssembly code, the function that wraps the JavaScript function call is imported as `$impFunc`. The function `$f0` is exported and calls the imported function `$impFunc` (lines 1-2). Note that the anonymous function used to wrap the original JavaScript function always has the same type signature, i.e., a void function with no parameters. Thus, the same WebAssembly module can be compiled once and reused for every replaced function call, changing only the import object containing the appropriate JavaScript function.

p_{T4a}. This transformation can be applied to locations where the call expressions do not have a return value that is used in an assignment or in another expression, as the primitive data types of WebAssembly, `i32`, `i64`, `f32`, `f64` cannot represent all possible JavaScript function return values.

The generalized variant *T4-CallExpression(b)* is a transformation rule $(L_{T4b}, p_{T4b}, t_{T4b})$ where:

L_{T4b}. L_{T4b} is the same as L_{T4a}.

t_{T4b}. t_{T4b} is defined in row T4-CallExpression(b) in Table I. The experimental WebAssembly Reference Types proposal [9] adds a new WebAssembly value type, `externref`, that can be used to pass references of arbitrary JavaScript values to and from WebAssembly. With this new type, the transformation only needs to import a reference to the transformed function. Specifically, the transformed JavaScript code only imports the function reference `f` into WebAssembly (line 2). In the original call expression, the function `f` is replaced with a WebAssembly export function `f0` that takes in the argument `a` of the original call and returns any value that the original function outputs (line 4). In the WebAssembly code, the export function `$f0` takes in a parameter of type `externref` and returns a value of type `externref` (line 1). Inside `$f0`, it calls the imported function `$impFunc` with a value that is the argument passed into `$f0` (lines 2-3). The benefits of this transformation over $T4_a$ include supporting more function calls by including those with return values and moving more behavior into WebAssembly than $T4_a$.

p_{T4b}. This variant of the call transformation can be applied only if: (i) the callee function does not contain a reference to `this`; (ii) the arguments of the call expression cannot contain a reference to `this`; (iii) the data dependencies of the variables referenced within the callee cannot contain `this`; (iv) the callee function is not a method of a literal value; and (v) the callee function is not the special functions `bind` or `super`. Conditions (i)-(iv) must be met because the value of `this` is changed when calling the function from within the WebAssembly module, which can lead to incorrect behavior. Condition (v) is needed as `bind` and `super` have restrictions on how they are called, so these functions cannot be passed as imports to the WebAssembly module.

5) Obfuscating If Statements: By leveraging the control-flow instructions in WebAssembly, the behavior of `if-else` statements in JavaScript can be moved to WebAssembly, removing the syntactic information while preserving the semantics. To this end, we present transformation rule *T5-IfStatement* (L_{T5}, p_{T5}, t_{T5}) where:

L_{T5}. The transformation rule applies to *IfStatement* nodes.

t_{T5}. The transformation is defined in row T5-IfStatement in Table I. At the translation site, t_{T5} wraps the code within the `if-` and `else`-blocks in two anonymous functions that are imported into the WebAssembly module (lines 2-4). A WebAssembly export function `f` is called and the result of the test condition of the original `if`-statement is converted to a (zero or one) integer that is passed as the argument to `f` (line 6). Within the WebAssembly module, the two functions wrapping the code within the `if-` and `else`-blocks are imported as `$imp1` and `$imp2` (lines 4,6). The exported function `$f` takes in an integer parameter which will be either zero or one to act as a Boolean (lines 1-7). `$f` contains `if-else` instructions that are decided by the function parameter `p`. If `p` is non-zero, the `if` instruction calls `$imp1` that contains the code originally in the `if`-block. Similarly, if `p` is zero, then the `else` instruction calls `$imp2` containing the code originally in the `else`-block. By leveraging the `if-else` instructions in WebAssembly, the original semantics are preserved while the use of the control-flow statement is hidden from the JavaScript syntax.

p_{T5}. The `if` statements can be transformed if the code blocks do not include the keywords `break`, `continue`, `return`, `yield`, or `throw`. These keywords are not compatible with moving the inner code blocks into functions.

6) Obfuscating For Loops: To obfuscate `for` loops, we define transformation rule *T6-ForStatement* (L_{T6}, p_{T6}, t_{T6}):

L_{T6}. The transformation locations where this rule applies are *ForStatement* nodes, which represent C-style `for` loops.

t_{T6}. The transformation is defined in row T6-ForStatement in Table I. At the translation site, t_{T6} wraps the loop condition, increment and body in three JavaScript functions that will be imported to the WebAssembly module (lines 3-8). The loop counter initialization can be hoisted out of the loop scope safely (line 2). Finally, a WebAssembly export function `f` is called which emulates the JavaScript `for` loop (line 11). In the WebAssembly module, the three JavaScript functions that are used to wrap the loop condition, increment, and body are imported as `$cond`, `$body`, and `$incre`, respectively (lines 4, 7, 8). The export function `$f` contains a `block` of code with label `$L0`, which encloses a `loop` block with label `$L1` (lines 2-3). Inside the `loop`, `$cond` is called (line 4) which evaluates the test condition of the JavaScript `for` loop and the result is checked using `i32.eqz` (line 5). If the test condition is false, the instruction `br_if $L0` branches out to the end of block `$L0`, terminating the loop (line 6). Otherwise, if the test condition is true, functions `$body` and `$incre` are called

to execute the statements within the loop body and the update expression (lines 7-8). The `br $L1` is used to branch to label `$L1`, which continues iteration (line 9).

p_{T6}. The precondition is the same as p_{T5}.

7) Obfuscating While Loops: Analogous to the above, we obfuscate `while` loops with a transformation rule *T7-WhileStatement* (L_{T7}, p_{T7}, t_{T7}) where:

L_{T7}. The rule applies to *WhileStatement* nodes.

t_{T7}. The transformation, defined in row T7-WhileStatement in Table I, is similar to t_{T6}. The only differences are that in t_{T7}, there is no loop increment nor loop counter initialization. A `while` loop can be created by only wrapping the loop condition and the loop body into anonymous functions and importing them to WebAssembly (lines 4, 7). The function `$f` is defined and exported, which emulates the JavaScript `while` loop by calling the import functions within the `loop` block.

p_{T7}. The precondition is the same as p_{T5} and p_{T6}.

C. Synchronous and Asynchronous WebAssembly Instantiation

For each transformation rule in Section IV-B (aside from T4-CallExpression(b)), we develop two variants differing in whether they instantiate the WebAssembly module synchronously or asynchronously, i.e., the implementation of the `instanWasm()` primitive.

```
1 let m = new WebAssembly.Module(
2   new Uint8Array(decodeBase64('...')));
3 return new WebAssembly.Instance(m, impObj);
```

Fig. 2. Synchronous WebAssembly instantiation.

The synchronous variants implement the primitive by using the `WebAssembly.Module` and `WebAssembly.Instance` constructor functions. Figure 2 shows the code, where in line 1, variable `m` is set to the compiled `WebAssembly.Module` object. The `WebAssembly.Module` constructor accepts a typed array containing the module bytes. Hence, we encode the module bytes into a base64 string, which is decoded to a typed `Uint8Array` at runtime (line 2). On line 3, the module object `m`, along with the import object, is then passed into the `WebAssembly.Instance` constructor. The returned instance object can be utilized by the transformation functions.

Since this is standard, synchronous code, there are no restrictions on how to integrate instantiation into the original JavaScript application. However, because the `WebAssembly.Module` constructor can block the JavaScript main thread, browser vendors discourage this method. Chromium in particular even limits the input module to at most 4KB in size [6] and throws an exception otherwise.

To get around this limitation, each synchronous transformation can emit one or more WebAssembly modules. Specifically, transformations T3-FunctionName, T4-CallExpression(a), T5-IfStatement, T6-ForStatement, and T7-WhileStatement only emit one WebAssembly module per file transformed. T1-StringLiteral, T2-ArrayInitialization, and T4-CallExpression(b) can emit one or more modules since the data stored within the modules can grow larger than 4KB. When a single WebAssembly module grows too large, e.g., because it contains many string literals in its data section, we split it into multiple modules to keep each under 4KB. For string literals larger than 4KB, the string literal is split across multiple modules and joined in the string reconstruction phase.

```
1 async function someFunction(){
2     // Transformation site: ...
3     await (async () => {
4     let mod = await WebAssembly.instantiateStreaming(
5       fetch("generated_module.wasm"), impObj);
6     //Transformation function code...
7     })()
8 }
```

Fig. 3. Instantiation of the asynchronous variant.

We also support asynchronous instantiation of the WebAssembly modules, which is the method that browser providers recommend. Asynchronous instantiation has several benefits, including unrestricted module size and the ability to put the generated WebAssembly modules in separate files. These benefits allow each transformation to only emit a single WebAssembly module.

For these variants, the `instanWasm()` primitive is implemented via the `WebAssembly.instantiateStreaming()` function, as shown in Figure 3 (line 4). This API spawns compilation on a separate thread, thus not blocking the main thread of execution. Since this API returns a *Promise*, we need to add the `await` keyword to allow the *Promise* to resolve before continuing. Line 6 represents a placeholder for the transformation code of T1-T7. The `await` keyword can only be employed in asynchronous functions, so we wrap the instantiation in an `async` anonymous function (line 3). Similarly, since the anonymous function is an `async` function, its invocation must also have the `await` keyword added (line 3). The enclosing function, `someFunction`, now contains the `await` keyword, so the function definition must have the `async` keyword added (line 1). Elsewhere in the code, any function calls to `someFunction` would also require adding the `await` keyword. As this example shows, inserting the `async`/`await` keywords to a translation site causes these keywords to be propagated to functions and call sites throughout the file.

This keyword propagation makes the asynchronous transformations non-trivial to design and implement. Specifically, we encountered three code locations that are difficult to propagate the `async`/`await` keywords to. First, anonymous functions used as parameters in other function calls, e.g., `.map`, are difficult to handle. Depending on the return type of the anonymous function, the `await` keyword may need to be added within the called function's definition or to the function invocation. Second, class constructors cannot be made `async`, so a check must be done to detect if a constructor is in the call chain of any function. Third, if a transformed function is exported from a module, any other files using the function as an import must be checked for functions and function calls to add `async` and `await` to.

All of the transformation rules have both synchronous and asynchronous variants except for T4-CallExpression(b). T4-CallExpression(b) relies on an experimental WebAssembly proposal that imposes complex preconditions. Adding

the asynchronous restrictions to this may break the original semantics and lead to incorrect transformations. T4-CallExpression(b) exposes more translation sites than T4-CallExpression(a), increasing the number of edge cases that can be encountered. We leave this combination as future work.

D. Applying Transformations

We now present the overall algorithm for applying these transformations to a given JavaScript AST. The input to the algorithm is a list of transformation rules and the AST of the original JavaScript file. The algorithm consists of three steps: (a) Identifying AST nodes where transformations should be applied, i.e., translation sites; (b) rewriting the AST by modifying the subtrees rooted at the translation sites; and (c) adding code to the AST root to instantiate the generated WebAssembly modules. The algorithm outputs the transformed AST corresponding to the obfuscated JavaScript code.

a) Identifying AST Nodes as Translation Sites: To identify translation sites, we perform a pre-order traversal of the AST starting at the root node. For each visited node n, the algorithm iterates through the list of transformation rules and checks which rules are applicable. A transformation rule (L, t, p) is applicable if the node n is in the set L of code locations and if the precondition p holds for n. A set of translation site nodes is produced for each transformation rule.

b) Rewriting AST Subtrees: After identifying all translation site nodes, the next step is to rewrite the subtrees rooted at these nodes. The algorithm applies transformations based on the size of the syntactic structures they target. Specifically, we iterate through the transformation rules in this order, applying each rule to all applicable subtrees before moving on to the next rule: T1-StringLiteral, T2-ArrayInitialization, T3-FunctionName, T4-CallExpression, T5-IfStatement, T6-ForStatement, T7-WhileStatement. This ordering ensures that transformations targeting finer-grained syntactic structures, such as string and array literals, are performed prior to transformations targeting coarser-grained structures, such as loops. If more coarse-grained transformation were applied first, the change could prevent more fine-grained transformations from being applied. For each rule (L, t, p), the algorithm visits all translation site nodes and applies the transformation function t, which modifies the AST in-place and yields a WebAssembly module used in the rewritten code. The output of this step is the rewritten AST and a set W of WebAssembly modules.

c) Adding WebAssembly Instantiation Code: The final step is adding code to instantiate the WebAssembly modules W. To this end, the algorithm inserts statements at the beginning of the script, i.e., at the root of the AST. For modules that are instantiated synchronously, we encode each module in W as a base64 string and add statements that decode and instantiate the modules. For asynchronously instantiated modules, we serialize the modules to separate files and issue corresponding `fetch` requests in the code. For asynchronous translations, the algorithm additionally adds the *async wrapper* (described in Section IV-A) around the root of the AST to support asynchronous keywords in the remainder of the code.

V. Implementation

We implement Wobfuscator with Node.js (v14.17.2) and TypeScript. The tool relies on the *Esprima* (v4.0.1) [3] and *Espree* (v7.3.1) [2] packages to parse the JavaScript files and on *Escodegen* (2.0.0) [1] to convert the transformed AST back into a JavaScript file. The *Wabt.js* (1.0.23) [8] package is used to generate the WebAssembly modules used in the obfuscation.

The data from our evaluation is available at https://github.com/js2wasm-obfuscator/translator. We make the implementation of Wobfuscator available upon request. We believe this strategy minimizes the threat of nefarious usage while also aiding researchers in independently reproducing our results, in confirming the identified weaknesses of existing malware detectors, and in serving as a foundation for future research on improving malware detectors.

VI. Evaluation

We evaluate Wobfuscator and its ability to obfuscate malicious JavaScript using opportunistic translation to WebAssembly along the following main research questions:

- **RQ1 – Effectiveness**: How effective is the approach at evading state-of-the-art JavaScript malware detectors and which transformations are most effective? How does our approach compare with other state-of-the-art obfuscators?
- **RQ2 – Correctness**: Do our code transformations preserve the semantics of the transformed code?
- **RQ3 – Efficiency**: How much runtime and code size overhead do the transformations impose, and how long does applying the transformations take?

To investigate these questions, we perform a comprehensive analysis on the effectiveness of our transformations in evading detection. We evaluate state-of-the-art detection tools against Wobfuscator on a large dataset of malicious and benign files. We evaluate the obfuscation advantage produced by Wobfuscator by comparing our approach against state-of-the-art open-source obfuscation tools. Lastly, we use the extensive test suites of widely used and mature *npm* modules to verify the correctness of our tool and demonstrate the runtime and code size overhead are acceptable for real-world usage.

A. Experimental Setup

1) Datasets: Due to different requirements, we use different datasets of JavaScript programs for different research questions. To answer RQ1, we need to train and apply state-of-the-art JavaScript malware detectors to large sets of real-world benign and malicious JavaScript code. Table II summarizes the datasets we use. The benign code consists of 149,677 files from the JS150k dataset [49]. The malicious code consists of 43,499 samples, with 2,674 samples from VirusTotal [7], 39,450 samples from the Hynek Petrak JavaScript malware collection [47], and 1,375 samples from the GeeksOnSecurity malicious JavaScript dataset [11]. These datasets are broken down further by the malware categories that they contain, such as trojans, ransomware, droppers. We list the breakdown of the malicious datasets in the first four columns of Table IX.

TABLE II
DATASETS FOR EVALUATING EFFECTIVENESS (RQ1).

Datasets		# Samples (Files)
Benign	JS150k	149,677
Malicious	VirusTotal	2,674
	Hynek Petrak	39,450
	GeeksOnSecurity	1,375
	Total Malicious	43,499

Answering RQ2 and RQ3 requires executing code before and after applying our transformations. We use a dataset of popular and large JavaScript projects on NPM with their test suites. To identify suitable projects, we select from the most depended-upon NPM modules [12] six modules that contain extensive test suites (first column of Table V).

2) JavaScript Malware Detectors: We evaluate our obfuscation technique against four state-of-the-art, static, learning-based JavaScript malware detectors. To train them, we split the benign and malicious datasets into training, validation, and test sets containing 70%, 15%, and 15% of the samples, respectively. We follow the steps provided by each project to train the detection models with the desired configuration.

Cujo [50] is a hybrid JavaScript malware detector that detects drive-by download attacks. It performs a lexical analysis of JavaScript files run on a website as well as a dynamic analysis by monitoring abstracted runtime behaviors. For our evaluation, we use the static detection part, based on a reimplementation of Cujo provided by Fass et al. [18].

Zozzle [24] is a mostly-static in-browser detection tool that uses syntactic information, such as identifier names and code locations, obtained from a JavaScript AST to identify malicious code. These features are input to a Bayesian classifier to label the samples as benign or malicious. We rely on a reimplementation of Zozzle provided by Fass et al. [19].

JaSt [28] is a static detector of malicious JavaScript that uses syntactic information from the AST to produce n-grams of sequential nodes to identify patterns indicative of malicious behavior. We use the implementation made available on the project's GitHub page [16].

JStap [27] is a static malware detector that leverages syntax, control-flow, and data-flow information by creating an AST, a Control Flow Graph (CFG), and a Program Dependency Graph (PDG), depending on the configuration. The tool extracts features either by constructing n-grams of nodes or by combining the AST node type with its corresponding identifier/literal value. In our evaluation, we focus on the PDG code abstraction with both the *n-grams* and *values* feature extraction modes. We use the implementation available on GitHub [17].

3) JavaScript Obfuscation Tools: We compare Wobfuscator against four open-source state-of-the-art JavaScript obfuscation tools. *JavaScript Obfuscator* [4] is a JavaScript obfuscation tool that supports multiple obfuscation techniques including variable renaming, dead code injection, and control-flow flattening. *Gnirts* [15] focuses on mangling string literals within JavaScript files. *Jfogs* [70] is an obfuscation tool that focuses on removing function call identifiers and parameters from call sites. *JSObfu* [5] is an obfuscator that supports converting function identifiers and string literals into expressions that evaluate to constants. This obfuscator also supports character escaping, whitespace removal, and more.

All the experiments on a desktop containing an Intel Core i7 CPU@3.20GHz w/ 32 GB of memory running Ubuntu 20.04.

B. Effectiveness in Evading Detection (RQ1)

1) Effectiveness of Our Approach: To evaluate the effectiveness of Wobfuscator at evading static malicious JavaScript detectors, we compare the detectors' performance on the original input programs against their performance after our obfuscation has been applied. Since the detectors classify each program as benign or malicious, the usual metrics of binary classifiers apply: precision and recall. Precision is the number of true positives (correctly identified malicious programs) divided by the number of all raised alarms (correct or not), and recall is the number of true positives divided by the number of all malicious programs in the dataset. That is $Prec=\frac{TP}{TP+FP}$, $Rec=\frac{TP}{TP+FN}$. A good malware detector should offer both high precision and high recall. Low precision indicates a high number of false positives, which would cause the system to block and break benign scripts and commonly used websites. Such a tool would not be adopted by actual users. Low recall means few of the actual malicious programs are detected, limiting the usefulness of the detector. The main goal of our obfuscation is to reduce the recall of detectors.

Some detectors fail to parse some of the original and transformed code samples due to outdated or incomplete support of the JavaScript language. Since this is an implementation-specific detail of these detectors rather than a result of their detection methodology, we choose to exclude these samples from the count rather than mark them as false negatives. As a result, the denominators of the recall results differ depending on the detector and the applied transformations.

Results: Table III shows the recall of the detectors described in Section VI-A2 (columns) when run on code obfuscated by our transformations (rows). The first row gives each detector's recall without our obfuscation, which serves as a baseline. The middle part of the table shows results from applying only one kind of transformation at a time. For example, the second row shows that applying our synchronous transformation technique T1-StringLiteral on the test set of malicious samples, Cujo achieves a recall of 0.61, i.e., a significant reduction compared to the baseline of 0.98. The results show that different translation techniques are more effective against some detectors rather than others. For each detector, the lowest recall score is bold-faced to reveal the best-performing individual transformation technique. For example, we find that T1-StringLiteral performs best for Cujo, Zozzle, and JStap in values mode, T4-CallExpression(a) performs best for JaSt, and T4-CallExpression(b) performs best for JStap in n-grams mode. Since each transformation rule is effective at reducing the recall for at least one detector, all transformation rules are integral to the effectiveness of our approach.

We explain the reasons why some detection tools disfavor certain transformation rules over others. Cujo performs a

TABLE III
RECALL OF MALWARE DETECTORS ON CODE OBFUSCATED BY WOBFUSCATOR. LOWEST RECALL IN **BOLD**.

Technique	Cujo	Zozzle	JaSt	JStap (NGrams)	JStap (Values)
Baseline: No transformation	0.98 (5,548/5,649)	0.66 (3,598/5,453)	0.99 (5,076/5,108)	0.99 (4,483/4,524)	0.98 (4,439/4,524)
Individual transformations:					
Sync, T1-StringLiteral	**0.61 (1,623/2,644)**	**0.62 (3,387/5,453)**	0.66 (3,393/5,108)	0.36 (1,539/4,257)	**0.43 (1,839/4,257)**
Sync, T2-ArrayInitialization	0.94 (4,050/4,292)	0.66 (3,593/5,450)	0.85 (4,360/5,105)	0.86 (3,890/4,505)	0.89 (4,009/4,505)
Sync, T3-FunctionName	0.67 (2,780/4,159)	0.65 (3,550/5,453)	0.69 (3,512/5,108)	0.57 (2,747/4,810)	0.72 (3,463/4,810)
Sync, T4-CallExpression(a)	0.71 (3,040/4,285)	0.64 (3,507/5,453)	**0.38 (1,943/5,108)**	0.37 (1,723/4,633)	0.78 (3,613/4,633)
Sync, T4-CallExpression(b)	0.58 (2,385/4,115)	0.63 (3,424/5,453)	0.44 (2,253/5,108)	**0.23 (1,058/4,586)**	0.73 (3,369/4,586)
Sync, T5-IfStatement	0.82 (3,513/4,301)	0.64 (3,505/5,453)	0.89 (4,535/5,108)	0.83 (3,717/4,501)	0.93 (4,178/4,501)
Sync, T6-ForStatement	0.90 (3,877/4,299)	0.66 (3,578/5,453)	0.92 (4,720/5,108)	0.87 (3,872/4,465)	0.98 (4,360/4,465)
Sync, T7-WhileStatement	0.90 (3,904/4,321)	0.66 (3,598/5,453)	0.96 (4,882/5,108)	0.98 (4,410/4,502)	0.98 (4,412/4,502)
Combined transformations:					
All sync (using T4(a))	**0.18 (416/2,255)**	**0.63 (3,450/5,450)**	0.22 (1,104/5,105)	**0.00 (1/4,235)**	0.18 (766/4,235)
All sync (using T4(b))	0.19 (415/2,205)	**0.63 (3,428/5,450)**	**0.18 (931/5,105)**	**0.00 (0/4,243)**	**0.08 (350/4,243)**
All async	0.28 (1,490/5,297)	0.65 (3,524/5,453)	0.20 (1,085/5,453)	**0.00 (4/4,612)**	0.22 (959/4,267)

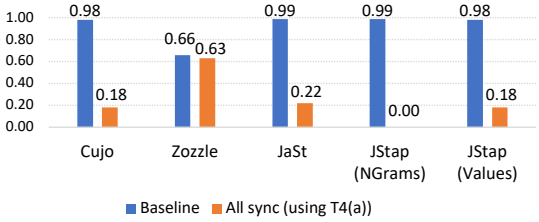

Fig. 4. Baseline recall vs. obfuscated recall.

lexical analysis on malware files, so it learns the suspicious features of strings that are indicative of malware. Since T1-StringLiteral removes these identifiers from the file, Cujo is not able to find suspicious tokens in the file. Zozzle identifies malicious combinations of syntax features and the context they are used in. Since T1-StringLiteral removes the usage of malicious strings, such as shell code and dynamic code, malicious files that have the majority of their code leveraging these strings will have indicative features removed from their AST, allowing them to evade Zozzle. JaSt uses n-grams of AST syntax nodes (with a length of four by default) to detect malicious patterns. T4-CallExpression(a) replaces a single *CallExpression* node with multiple nodes handling the WebAssembly module instantiation. Since this translation adds more nodes than the length of sliding window, it can throw off the detection of known malicious n-gram patterns. JStap in its n-grams mode generates a program dependency graph (PDG) by adding edges representing data flow to the AST of an input file. Since T4-CallExpression(b) (and other transformations) route data flow through the generated WebAssembly module, it hinders the n-gram features that JStap can extract when performing its data flow analysis. JStap in its values mode is most affected by the T1-StringLiteral transformation since, in this mode, it relies on literals when selecting features to extract. T1-StringLiteral removes some literals.

The lower part of Table III shows results from applying all transformation rules at once. We report results for three transformation combinations. "All sync (using T4(a))" and "All sync (using T4(b))" mean all synchronous transformations with T4(a) being used and with T4(b) being used, respectively. "All async" means all transformations in Table I (aside from T4(b)) in their asynchronous variant. We find that combining all transformation rules greatly reduces the recall of the detectors. In particular, with the "All sync (using T4(a))" set of transformations, Cujo, Zozzle, JaSt, JStap (NGrams), and JStap (Values) have a recall of 0.18, 0.63, 0.22, 0.00, and 0.18, respectively. Because of its performance and compatibility with the WebAssembly MVP language, we select "All sync (using T4(a))" to be the default configuration for Wobfuscator. Figure 4 visualizes the results for "All sync (using T4(a))".

The main goal of our work is to reduce the recall of detectors, but we also measure their precision. The precision values for the transformations are listed in Table VII. For most applied transformations, the precision remains between 0.9 and 1.0. However, certain transformations can greatly impact the precision on some detectors, e.g., "All sync (using T4(a))" reduces JStap (NGrams) precision to 0.5. This shows that while reducing the precision is not its main objective, Wobfuscator can reduce the precision of certain detectors.

2) Comparison with Other Obfuscators: To demonstrate how Wobfuscator compares against currently available JavaScript obfuscators, we evaluate four obfuscation tools on the same dataset used in Section VI-B1. We collect the precision and recall values obtained by the five malware detection tools when evaluated on a dataset obfuscated by each tool. Similar to Section VI-B1, some detectors fail to parse certain obfuscated files, leading to different denominators in the values within the same detector column.

Results: Table IV shows the recall values of the detection tools (columns) when run on code obfuscated by each of the four obfuscation tools described in Section VI-A3 (rows). The last row shows the best recall values obtained by Wobfuscator.

The results show that Wobfuscator outperforms current obfuscators when compared on the recall reduction of malware detectors. The only exception occurs when Jfogs is evaluated against JaSt. In this case, Jfogs' recall rate of 0.00 outperforms Wobfuscator's recall rate of 0.18. Jfogs' obfuscation primarily replaces identifiers and literals with new intermediate variables, so Wobfuscator could be used to compliment Jfogs. For example, Jfogs moves string literals into variables, but

TABLE IV
RECALL OF MALWARE DETECTORS ON CODE OBFUSCATED BY WOBFUSCATOR AND OTHER OBFUSCATORS.

Obfuscator	Cujo	Zozzle	JaSt	JStap (NGrams)	JStap (Values)
JavaScript Obfuscator	1.00 (4,406/4,415)	0.70 (3,807/5,453)	0.81 (4,153/5,108)	0.43 (2,005/4,717)	0.62 (2,947/4,717)
Gnirts	0.98 (5,548/5,649)	0.66 (3,598/5,453)	0.99 (5,076/5,108)	0.99 (4,483/4,524)	0.98 (4,439/4,524)
Jfogs	0.77 (3,515/4,562)	0.66 (3,584/5,453)	0.00 (26/5,453)	0.00 (16/5,025)	0.56 (2,826/5,025)
JSObfu	1.00 (4,994/5,008)	0.84 (4,467/5,324)	0.29 (1,456/4,979)	0.01 (20/3,667)	0.66 (2,420/3,667)
Wobfuscator (best recall)	0.18 (416/2,255)	0.62 (3,387/5,453)	0.18 (931/5,105)	0.00 (4/4,612)	0.08 (350/4,243)

it does not alter or remove the strings from the file. Using the T1-StringLiteral transformation, the string literals can be completely removed from the JavaScript file, reducing the syntactic information available to the detectors.

3) Breakdown of Results by Malware Type: The malicious datasets contain several different categories of malware, including cryptominers, trojans, and droppers. We provide a breakdown on the malware categories contained within our dataset. VirusTotal provides the malware type reported by the AV scanners. We reduce the number of malware categories presented by merging similar groups together, e.g., merging *JS:Trojan.Gnaeus* and *JS:Trojan.Agent* into a *Trojan* category. For GeeksOnSecurity, we use the directory names to identify which samples are exploits kits and which are JavaScript droppers. The Hynek Petrak dataset does not provide metadata on the samples, so we scan the files with ClamAV to obtain the malware categories. To demonstrate the effectiveness of Wobfuscator in obfuscating a diverse set of malware samples, we measure the reduction in the recall of malware detectors for different malware categories in the dataset. For space reasons, we list only the minimum recall rates observed within the malware categories among all of the transformations.

Results: Columns 5-9 of Table IX present the recall values obtained by the malware detectors (columns) when samples from each malware category (rows) are obfuscated. In addition to the recall rate, each cell lists the number of files correctly marked as malicious by the detector over the number of malicious files the detector tested within that category. Columns marked with '-' signify that the detectors are unable to parse any of the test files within the malware category. For the Phishing malware category, no samples appear in our test dataset, so no recall values are available. The results on the five largest malware groups (Downloader, Misc., Trojan, Malware, and Exploit) show that Wobfuscator can significantly reduce recall rates across diverse malware categories.

C. Correctness of the Transformations (RQ2)

The transformations we apply change the syntactic structure of the program. Naturally, such changes could affect program semantics, potentially making the obfuscated program behave differently from the original program and thus breaking functional correctness. This gives rise to two questions, which are in tension with each other: First, do we preserve functional correctness of the input program, i.e., are our code transformations semantics-preserving? Second, how often are the transformations applied? To validate that the correctness of the program is preserved, we leverage the comprehensive test suites of existing widely used JavaScript projects. We apply the transformations to the tested code and then validate if the transformed code still passes its tests. Addressing the second question, a trivial solution to correctness would be to transform only a very small set of code locations, preserving semantics at the expense of obfuscating less code. Thus, we also evaluate how often each transformation rule is applied.

The validation setup differs between the synchronous and asynchronous variants of the transformations. In the synchronous case, we can simply apply the transformations to any existing code. To validate the correctness of the asynchronous transformations, the projects must be modified to support asynchronous execution as described in Section IV-C. Since automatically turning arbitrary JavaScript code into asynchronous code is non-trivial, we instead focus on an NPM project, *node-fetch*, that is already asynchronous, so applying the asynchronous transformations is simplified. We use this project to validate the asynchronous variants and use the other five projects to validate the synchronous variants.

Results: The results for the test suite runs are shown in Table V. This table lists the tested project, its version, and the number of translation sites where rules T1–T7 are applied to. The last two columns list the total number of tests in the test suite, and the number of tests that are impacted by at least one transformation. All tests in each project pass successfully, showing that our obfuscations are semantics-preserving.

Columns 4-11 of Table V show the number of transformed code locations that meet the preconditions of the transformation out of the total number of available code locations relevant to the transformation, regardless of whether they satisfy the preconditions. The last two columns of Table V show that of the 2,017 unit tests in the five test suites, 1,844 of them (91.42%) rely on a function that is impacted by at least one transformation rule. The results show that our transformation rules are applicable to code locations used in the real-world.

D. Efficiency in Terms of Runtime and Code Size (RQ3)

The Wobfuscator transformations we propose re-implement native JavaScript functionalities in WebAssembly modules, such as calling a function, performing a while loop, initializing an array, etc. As a result, there will be an impact on the performance of the translated programs. To quantify the performance impact of our transformations, we use the test suites of the six modules described in Section VI-C. In addition, the code size increase caused by the transformations is also analyzed, counting both added JavaScript and WebAssembly code.

1) Translation Runtime: First, we measure the time taken for performing the transformations on the project files. The times are measured with the `time` command available in

TABLE V
CORRECTNESS VALIDATION RESULTS.

Project	Version	LoC	T1	T2	T3	T4(a)	T4(b)	T5	T6	T7	Total # of Tests	# of Tests Impacted
Validation of synchronous transformations:												
Lodash	5.0.0	21,178	193/801	160/208	0/0	87/467	299/467	47/187	0/0	38/61	408	322
Chalk	4.1.0	319	64/68	0/23	2/2	21/108	26/108	4/25	1/1	2/2	54	54
Commander	7.2.0	1,153	155/163	0/28	0/0	130/394	87/394	91/153	4/4	2/3	632	592
Debug	4.3.2	505	141/149	2/8	0/0	20/95	43/95	16/29	2/5	0/0	14	13
Async	3.2.0	787	30/57	0/21	0/0	89/209	108/209	31/86	5/6	5/8	675	659
Validation of asynchronous transformations:												
Node-Fetch	3.0.0beta.10	970	174/212	1/17	0/0	49/264	-	26/94	0/0	0/0	234	204
Total	-	4,152	757/1,450	163/305	2/2	396/1,537	654/1,537	189/480	12/16	47/74	2,017	1,844

TABLE VI
EFFICIENCY OF TRANSFORMATIONS.

Project	Translation time		Execution time		Code size (bytes)	
	LoC	Time	Original	Overhead	Original	Overhead
Synchronous Validation						
Lodash	21,178	29.58s	3.51s	+25.81%	135,402	+139.84%
Chalk	319	0.81s	4.03s	+7.01%	14,935	+166.70%
Commander	1,153	0.51s	3.97s	+49.95%	74,269	+146.96%
Debug	505	1.14s	0.61s	+3.24%	21,395	+154.70%
Async	787	5.42s	16.83s	+2079.21%	28,925	+363.52%
Asynchronous Validation						
Node-Fetch	970	16.03s	4.11s	+14.76%	51,960	127.16%
Average	4,152	8.92s	5.68s	+31.07%	54,481	+170.42%

Linux, averaged over ten repetitions. We compute the total transformation time of a project by summing the times to convert each JavaScript file used in the project. The transformation time results are presented in Table VI. The table shows that for the largest project, Lodash with 21,178 lines of code, the average time to apply all of the synchronous transformations is 29.58 seconds. For the smallest project, Chalk with 319 lines of code, the average time to apply the transformations is only 0.81 seconds. In addition, we find that among all the projects in Table VI, the average time to apply all of the transformations is only 8.92 seconds. These low transformation times show that Wobfuscator is practical for JavaScript obfuscation.

2) Execution Time Overhead: The execution overhead time is the increase in runtime to complete the execution of the test suites of the transformed projects. We use the `time` command to measure the runtime of the project test suite before and after the transformations are applied, reporting averages over ten repeated measurements. The execution time results are presented in Table VI. Our transformations add a performance overhead that ranges from an increase of 3.24% to an increase of 2,079%. While the highest overhead number is large, it is important to note that this large runtime originates from one test within the *async* project that concurrently applies an asynchronous function to a collection of 1,048,576 elements. In most cases, it is unlikely that malware samples will follow such an execution pattern that incurs this large overhead. On average, Wobfuscator adds a performance overhead of 31.07%.

3) Code Size Overhead: The code size overhead is the increase in code size between the original file and transformed output among all code files within a project. Table VI lists the percentage of growth in code size compared with the original size. On average, applying all of the transformations among the project, the code size increased by 170.42%. Overall, the code size overhead is acceptable for practical applications.

VII. DISCUSSION

This section discusses the limitations and possible mitigations to defend against WebAssembly-based obfuscation.

A. Limitations

Wobfuscator targets malware detectors based on static analysis, and despite its effectiveness in bypassing them, is unlikely to be equally effective for dynamic analysis-based detectors. The transformations move some behavior into WebAssembly while leaving the ultimate runtime behavior intact. That is, a dynamic detector that, e.g., observes browser API calls made by a website will observe the same behavior with and without our obfuscation. However, in practice static detectors are much easier to deploy (e.g., as network proxies or browser extensions), whereas observing dynamic behavior is more complex to set up and expensive at runtime.

Another limitation is that the approach applies transformations only to some of the given code. If a code location does not fulfill the preconditions for a specific transformation, then it cannot be transformed. Conservatively guarding transformations is crucial to ensure that our approach preserves the semantics of the given code, but also limits its applicability.

Finally, our obfuscation relies on WebAssembly being available in the browser. With WebAssembly support in 94% of all installed browsers, this limitation is likely to be acceptable in practice. To ensure that the obfuscated malware runs as expected, an attacker could check for WebAssembly support and load the obfuscated code only if the language is supported.

B. Mitigations

We discuss three mitigation strategies aimed at detecting malware despite our obfuscation. The first is dynamic analysis-based malware detection. Because our approach preserves the original JavaScript behavior, many runtime characteristics that dynamic detectors focus on [53] are not affected by the obfuscation. WebAssembly code invokes web APIs through JavaScript, which means the call will be visible to any runtime analysis that wraps the API functions or intercepts them within the browser. However, dynamic malware detectors often impose a non-negligible runtime overhead and may miss malware that hides its malicious behavior in specific configurations.

The second mitigation strategy is based on the defender knowing the details of our obfuscation. Since the WebAssembly usage in the obfuscated code, e.g., loading many small modules, may be abnormal, it is possible to define heuristic rules to detect that Wobfuscator was applied. In a similar vein, one could include code obfuscated with our technique in the training data used to learn a malware classifier. The main drawback of these mitigations is that obfuscation does not imply maliciousness. There are legitimate reasons for obfuscating code, e.g., protecting intellectual property. Hence, classifying all code obfuscated by our technique, or any other obfuscation technique, as malicious is likely to cause an unacceptably high number of false positives.

Finally, the third mitigation strategy is to jointly analyze JavaScript and WebAssembly. For detectors based on traditional program analysis, both static or dynamic, a joint analysis would reason about how data and control flows between the two languages. Likewise, learning-based detectors, such as those used in our evaluation, could feed code in both languages into their models. We are not aware of any existing malware detector with support for WebAssembly but hope that our work will raise awareness that a joint analysis would be useful.

VIII. Related Work

Obfuscation Studies and Techniques: Obfuscation techniques have been observed in various programming languages and software domains for both malicious and benign purposes. Previous works categorize obfuscation techniques applied on malicious code [69] while others compare the effectiveness of different obfuscation techniques [22], [32], [61]. Some studies have analyzed the usage of obfuscation techniques specifically in JavaScript code by investigating the obfuscation techniques used in real-world malicious and benign files [66], [55].

There is little work proposing new obfuscation attacks for JavaScript code. Fass et al. [26] construct HideNoSeek which rewrites the ASTs of malicious programs into the AST of known benign programs to avoid detection. The authors evaluate HideNoSeek on 91,020 samples against VirusTotal, Yara, JaSt, Zozzle and Cujo. The obfuscation technique is able to achieve a 99.98% false negative rate against the detectors.

Malware Detection: An active area in academic research produces static analysis techniques designed to identify malicious behavior even in the presence of obfuscated code.

One class of static detection tools use lexical and syntactic information derived from JavaScript files in order to identify features that indicate malicious code [50], [24], [16], [20], [54]. Techniques build on this syntactic information by incorporating control-flow and data-flow analysis [27] or by adding dynamic analysis to confirm the presence of malware [67], [59]. Other static detection techniques analyze the JavaScript source code through machine-learning and deep-learning approaches [63], [46]. Wobfuscator can impact the detection rates of these detectors since it reduces the syntactic information available. In addition, some behavior is moved to WebAssembly modules, which are ignored by these detectors.

Other malware detection techniques dynamically analyze programs to identify malicious behaviors. Some techniques focus on collecting runtime statistics to construct models that identify malware [23], [53], [68]. Other techniques leverage symbolic execution [39] or forced execution [38] to trigger malware hidden behind complex input sequences. Wobfuscator is unlikely to reduce the detection rate against these detectors as we do not significantly change the runtime behavior.

Obfuscation Detection: Some existing work only focuses on detecting obfuscation rather than obfuscated malware. NOFUS [35] and JSOD [13] use syntactic and contextual information as features for a machine learning classifier to detect obfuscation. Sarker et al. [53] develop a hybrid approach by instrumenting browser APIs and determining whether the traced API call corresponds to a static code location.

WebAssembly and WebAssembly Security: Haas et al. [31] explain the motivation and benefits of introducing a new byte code language to the Web. Several works investigate security aspects of WebAssembly, e.g., unsolicited cryptomining in the browser [40], [45], [52] and how to detect and defend against it [37], [51], [62]. Lehmann et al. show that source-level memory vulnerabilities may propagate to WebAssembly binaries [41], a problem that affects many real-world binaries [33]. However, neither of those works use WebAssembly to hide arbitrary JavaScript behavior from inspection. Future joint malicious code detectors for JavaScript and WebAssembly could build upon the Wasabi framework [42] or taint tracking frameworks for WebAssembly [29], [57].

IX. Conclusion

Much work has focused on identifying JavaScript malware using static analysis. However, these techniques ignore recent web standards available to attackers, namely WebAssembly. To bypass static detectors, we present Wobfuscator, an obfuscation approach built on a set of seven transformation rules that opportunistically translate specific parts of JavaScript code into functionally identical WebAssembly modules. We evaluate our transformations against four state-of-the-art static JavaScript malware detectors and show that our approach effectively reduces the recall on real malware samples. We show that our technique outperforms other obfuscation tools only based on JavaScript. Finally, we use the test suites of six NPM packages to validate the correctness of our transformations and show their low performance overhead. Our results show that current static detectors are ineffective against techniques that implement cross-language code obfuscation, motivating future work on addressing this challenge.

X. Acknowledgments

We thank the anonymous reviewers and the shepherd for their constructive comments. This work was partially supported by the US National Science Foundation under Grant No. 2047980, the European Research Council (ERC, grant agreement 851895), and the German Research Foundation within the ConcSys and Perf4JS projects. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] "escodegen." [Online]. Available: https://www.npmjs.com/package/escodegen

[2] "espree." [Online]. Available: https://www.npmjs.com/package/espree

[3] "Esprima." [Online]. Available: https://esprima.org/

[4] GitHub - javascript-obfuscator/javascript-obfuscator: A powerful obfuscator for JavaScript and Node.js. [Online]. Available: https://github.com/javascript-obfuscator/javascript-obfuscator

[5] "JSObfu," Rapid7. [Online]. Available: https://github.com/rapid7/jsobfu

[6] "Loading WebAssembly modules efficiently." [Online]. Available: https://developers.google.com/web/updates/2018/04/loading-wasm

[7] "VirusTotal." [Online]. Available: https://www.virustotal.com/gui/home/upload

[8] "wabt." [Online]. Available: https://www.npmjs.com/package/wabt

[9] "WebAssembly Core Specification," https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[10] "WebAssembly Website." [Online]. Available: https://webassembly.org/

[11] "Malicious Javascript Dataset," Aug. 2021, original-date: 2017-01-31T17:48:24Z. [Online]. Available: https://github.com/geeksonsecurity/js-malicious-dataset

[12] 262588213843476, "npm rank." [Online]. Available: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

[13] I. A. AL-Taharwa, H.-M. Lee, A. B. Jeng, K.-P. Wu, C.-S. Ho, and S.-M. Chen, "JSOD: JavaScript obfuscation detector: JSOD," *Security and Communication Networks*, vol. 8, no. 6, pp. 1092–1107, Apr. 2015. [Online]. Available: https://doi.org/10.1002/sec.1064

[14] M. Alsharnouby, F. Alaca, and S. Chiasson, "Why phishing still works: User strategies for combating phishing attacks," *Int. J. Hum. Comput. Stud.*, vol. 82, pp. 69–82, 2015. [Online]. Available: https://doi.org/10.1016/j.ijhcs.2015.05.005

[15] anseki, "Gnirts." [Online]. Available: https://github.com/anseki/gnirts

[16] Aurore54F, "JaSt - JS AST-Based Analysis," Jun. 2021, original-date: 2017-04-10T19:03:18Z. [Online]. Available: https://github.com/Aurore54F/JaSt

[17] ——, "JStap: A Static Pre-Filter for Malicious JavaScript Detection," Jun. 2021, original-date: 2019-09-02T13:44:26Z. [Online]. Available: https://github.com/Aurore54F/JStap

[18] ——, "lexical-jsdetector," May 2021, original-date: 2019-09-20T05:52:48Z. [Online]. Available: https://github.com/Aurore54F/lexical-jsdetector

[19] ——, "syntactic-jsdetector," Jun. 2021, original-date: 2019-09-20T05:54:00Z. [Online]. Available: https://github.com/Aurore54F/syntactic-jsdetector

[20] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th international conference on World wide web - WWW '11*. Hyderabad, India: ACM Press, 2011, p. 197. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1963405.1963436

[21] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362800

[22] M. Ceccato, A. Capiluppi, P. Falcarin, and C. Boldyreff, "A Large Study on the Effect of Code Obfuscation on the Quality of Java Code," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1486–1524, Dec. 2015. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9321-0

[23] M. Cova, C. Krügel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 281–290. [Online]. Available: https://doi.org/10.1145/1772690.1772720

[24] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, "ZOZZLE: fast and precise in-browser javascript malware detection," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full_papers/Curtsinger.pdf

[25] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1388–1401. [Online]. Available: https://doi.org/10.1145/2976749.2978313

[26] A. Fass, M. Backes, and B. Stock, "HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1899–1913. [Online]. Available: https://doi.org/10.1145/3319535.3345656

[27] ——, "JStap: a static pre-filter for malicious JavaScript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, D. Balenson, Ed. ACM, 2019, pp. 257–269. [Online]. Available: https://doi.org/10.1145/3359789.3359813

[28] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Giuffrida, S. Bardin, and G. Blanc, Eds., vol. 10885. Springer, 2018, pp. 303–325. [Online]. Available: https://doi.org/10.1007/978-3-319-93411-2_14

[29] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *CoRR*, vol. abs/1802.01050, 2018. [Online]. Available: http://arxiv.org/abs/1802.01050

[30] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, P. Champin, F. Gandon, M. Lalmas, and P. G. Ipeirotis, Eds. ACM, 2018, pp. 309–318. [Online]. Available: https://doi.org/10.1145/3178876.3186097

[31] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 185–200.

[32] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 421–431. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3180228

[33] A. Hilbig, D. Lehmann, and M. Pradel, "An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, Eds. ACM / IW3C2, 2021, pp. 2696–2708. [Online]. Available: https://doi.org/10.1145/3442381.3450138

[34] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1701–1713. [Online]. Available: https://doi.org/10.1145/3243734.3243840

[35] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Cursinger, ""NOFUS: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"," Tech. Rep. MSR-TR-2011-57, May 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/nofus-automatically-detecting-string-fromcharcode32-obfuscated-tolowercase-javascript-code/

[36] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 637–652. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/kapravelos

[37] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, 2019, pp. 840–852.

[38] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-Force: Forced Execution on JavaScript," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, Apr. 2017, pp. 897–906. [Online]. Available: https://doi.org/10.1145/3038912.3052674

[39] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking Internet Malware," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 443–457, iSSN: 2375-1207.

[40] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *CCS*, 2018.

[41] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 217–234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[42] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing WebAssembly," in *ASPLOS*, 2019.

[43] L. Lu, V. Yegneswaran, P. A. Porras, and W. Lee, "BLADE: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010, pp. 440–450. [Online]. Available: https://doi.org/10.1145/1866307.1866356

[44] Microsoft, "Microsoft digital defense report," 2020.

[45] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *DIMVA*, 2019.

[46] S. Ndichu, S. Kim, S. Ozawa, T. Misu, and K. Makishima, "A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors," *Applied Soft Computing*, vol. 84, p. 105721, Nov. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1568494619305022

[47] H. Petrak, "Javascript Malware Collection," Aug. 2021, original-date: 2017-05-07T19:17:23Z. [Online]. Available: https://github.com/HynekPetrak/javascript-malware-collection

[48] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser: Analysis of web-based malware," in *First Workshop on Hot Topics in Understanding Botnets, HotBots'07, Cambridge, MA, USA, April 10, 2007*, N. Provos, Ed. USENIX Association, 2007. [Online]. Available: https://www.usenix.org/conference/hotbots-07/ghost-browser-analysis-web-based-malware

[49] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," *SIGPLAN Not.*, vol. 51, no. 1, p. 761–774, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837671

[50] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, C. Gates, M. Franz, and J. P. McDermott, Eds. ACM, 2010, pp. 31–39. [Online]. Available: https://doi.org/10.1145/1920261.1920267

[51] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1129–1140. [Online]. Available: https://doi.org/10.1145/3324884.3416580

[52] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld, "Digging into browser-based crypto mining," in *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*. ACM, 2018, pp. 70–76. [Online]. Available: https://dl.acm.org/citation.cfm?id=3278539

[53] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage," in *IMC '20: ACM Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*. ACM, 2020, pp. 648–661. [Online]. Available: https://doi.org/10.1145/3419394.3423616

[54] P. Seshagiri, A. Vazhayil, and P. Sriram, "AMA: Static Code Analysis of Web Page for the Detection of Malicious Scripts," *Procedia Computer Science*, vol. 93, pp. 768–773, 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S187705091631537X

[55] P. Skolka, C. Staicu, and M. Pradel, "Anything to Hide? Studying Minified and Obfuscated Code in the Web," in *The World Wide Web Conference (WWW)*. ACM, 2019, pp. 1735–1746. [Online]. Available: https://doi.org/10.1145/3308558.3313752

[56] B. Stock, B. Livshits, and B. G. Zorn, "Kizzle: A signature compiler for detecting exploit kits," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 2016, pp. 455–466. [Online]. Available: https://doi.org/10.1109/DSN.2016.48

[57] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," *CoRR*, vol. abs/1807.08349, 2018. [Online]. Available: http://arxiv.org/abs/1807.08349

[58] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen, "Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets," in *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, S. Moriai, T. Jaeger, and K. Sakurai, Eds. ACM, 2014, pp. 525–530. [Online]. Available: https://doi.org/10.1145/2590296.2590311

[59] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, "JSDC: A Hybrid Approach for JavaScript Malware Detection and Classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. Singapore Republic of Singapore: ACM, Apr. 2015, pp. 109–120. [Online]. Available: https://dl.acm.org/doi/10.1145/2714576.2714620

[60] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, "Translingual obfuscation," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 128–144. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.21

[61] P. Wang, D. Wu, Z. Chen, and T. Wei, "Field experience with obfuscating million-user ios apps in large enterprise mobile development," *Software: Practice and Experience*, vol. 49, no. 2, pp. 252–273, 2019.

[62] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 122–142.

[63] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious JavaScript code: Using a deep learning approach to detect JavaScript-based attacks," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, Jul. 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/sec.1441

[64] WebAssembly Community Group, "WebAssembly Specification," 2017. [Online]. Available: https://webassembly.github.io/spec/core/

[65] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, A. Gangemi, S. Leonardi, and A. Panconesi, Eds. ACM, 2015, pp. 1286–1295. [Online]. Available: https://doi.org/10.1145/2736277.2741630

[66] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*. IEEE Computer Society, 2012, pp. 9–16. [Online]. Available: https://doi.org/10.1109/MALWARE.2012.6461002

[67] ——, "JStill: mostly static detection of obfuscated malicious JavaScript code," in *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*. San Antonio, Texas, USA: ACM Press, 2013, p. 117. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2435349.2435364

[68] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan, "Detection and classification of malicious JavaScript via attack behavior modelling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, Jul. 2015, pp. 48–59. [Online]. Available: https://doi.org/10.1145/2771783.2771814

[69] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.

[70] Zswang, "Zswang/jfogs." [Online]. Available: https://github.com/zswang/jfogs

APPENDIX

TABLE VII
PRECISION OF MALWARE DETECTORS ON CODE OBFUSCATED BY WOBFUSCATOR.

Technique	Cujo	Zozzle	JaSt	JStap (NGrams)	JStap (Values)
Individual transformations:					
Baseline (no transformation)	0.95 (5,548/5,832)	0.97 (3,598/3,694)	1.00 (5,076/5,080)	1.00 (4,483/4,484)	1.00 (4,439/4,440)
Sync, T1-StringLiteral	0.85 (1,623/1,907)	0.97 (3,387/3,483)	1.00 (3,393/3,397)	1.00 (1,539/1,540)	1.00 (1,839/1,840)
Sync, T2-ArrayInitialization	0.93 (4,050/4,334)	0.97 (3,593/3,689)	1.00 (4,360/4,364)	1.00 (3,890/3,891)	1.00 (4,009/4,010)
Sync, T3-FunctionName	0.91 (2,780/3,064)	0.97 (3,550/3,646)	1.00 (3,512/3,516)	1.00 (2,747/2,748)	1.00 (3,463/3,464)
Sync, T4-CallExpression(a)	0.91 (3,040/3,324)	0.97 (3,507/3,603)	1.00 (1,943/1,947)	1.00 (1,723/1,724)	1.00 (3,613/3,614)
Sync, T4-CallExpression(b)	0.89 (2,385/2,669)	0.97 (3,424/3,520)	1.00 (2,253/2,257)	1.00 (1,058/1,059)	1.00 (3,369/3,370)
Sync, T5-IfStatement	0.93 (3,513/3,797)	0.97 (3,505/3,601)	1.00 (4,535/4,539)	1.00 (3,717/3,718)	1.00 (4,178/4,179)
Sync, T6-ForStatement	0.93 (3,877/4,161)	0.97 (3,578/3,674)	1.00 (4,720/4,724)	1.00 (3,872/3,873)	1.00 (4,360/4,361)
Sync, T7-WhileStatement	0.93 (3,904/4,188)	0.97 (3,598/3,694)	1.00 (4,882/4,886)	1.00 (4,410/4,411)	1.00 (4,412/4,413)
Combined transformations:					
All sync (using T4(a))	0.59 (416/700)	0.97 (3,450/3,546)	1.00 (1,104/1,108)	0.50 (1/2)	1.00 (766/767)
All sync (using T4(b))	0.59 (415/699)	0.97 (3,428/3,524)	1.00 (931/935)	0.00 (0/1)	1.00 (350/351)
All async	0.84 (1,490/1,774)	0.97 (3,524/3,620)	1.00 (1,085/1,089)	0.80 (4/5)	1.00 (959/960)

TABLE VIII
PRECISION OF MALWARE DETECTORS ON CODE OBFUSCATED BY OTHER OBFUSCATORS.

Obfuscator	Cujo	Zozzle	JaSt	JStap (NGrams)	JStap (Values)
javascript-obfuscator	0.94 (4,406/4,690)	0.98 (3,807/3,903)	1.00 (4,153/4,157)	1.00 (2,005/2,006)	1.00 (2,947/2,948)
Gnirts	0.95 (5,548/5,832)	0.97 (3,598/3,694)	1.00 (5,076/5,080)	1.00 (4,483/4,484)	1.00 (4,439/4,440)
jfogs	0.93 (3,515/3,799)	0.97 (3,584/3,680)	0.87 (26/30)	0.94 (16/17)	1.00 (2,826/2,827)
JSObfu	0.95 (4,994/5,278)	0.98 (4,467/4,563)	1.00 (1,456/1,460)	0.95 (20/21)	1.00 (2,420/2,421)

TABLE IX
MALWARE CATEGORY BREAKDOWN OF DATASETS AND DETECTOR RECALL RATES.

Malware Type	Dataset Breakdown			Malware Detector Recall				
	VirusTotal	GeeksOnSecurity	HynekPetrak	Cujo	Zozzle	JaSt	JStap (N-grams)	JStap (Values)
Downloader	14	0	21619	0.10(102/988)	0.67(1772/2629)	0.11(279/2629)	0.00(0/2440)	0.00(2/1787)
Misc	167	21	13527	0.35(257/731)	0.64(1100/1732)	0.36(625/1732)	0.00(0/1082)	0.06(60/1082)
Trojan	1618	0	2058	0.00(0/201)	0.61(305/499)	0.00(0/499)	0.00(0/453)	0.06(25/453)
Malware	0	0	1546	0.00(0/109)	0.29(66/231)	0.22(51/231)	0.00(0/180)	0.02(3/125)
Exploit	6	1029	0	0.00(0/151)	0.00(0/151)	0.00(0/151)	0.00(0/151)	0.00(0/151)
Ransomware	12	0	700	0.00(0/56)	0.05(3/59)	0.00(0/3)	0.00(0/56)	-
Cryptominer	665	0	0	1.00(1/1)	0.99(68/69)	0.00(0/69)	0.00(0/1)	0.00(0/1)
Dropper	0	323	0	0.00(0/17)	0.50(26/45)	0.11(5/45)	0.00(0/45)	0.00(0/45)
Hijacker	112	0	0	0.00(0/12)	0.94(17/18)	0.00(0/18)	0.00(0/12)	0.00(0/11)
Riskware	21	0	0	1.00(1/1)	1.00(5/5)	0.00(0/5)	-	-
Redirector	15	0	0	0.00(0/3)	0.25(1/4)	0.00(0/4)	0.00(0/3)	0.00(0/3)
Clicker	25	0	0	-	1.00(4/4)	0.00(0/4)	-	-
Iframe	24	0	0	-	1.00(4/4)	0.00(0/4)	1.00(1/1)	1.00(1/1)
Clickjack	13	0	0	-	1.00(1/1)	0.00(0/1)	-	-
Phishing	3	0	0	-	-	-	-	-