Static Semantics Reconstruction for Enhancing JavaScript-WebAssembly Multilingual Malware Detection

Yifan Xia, Ping He, Xuhong Zhang(✉), Peiyu Liu, Shouling Ji, and Wenhai Wang(✉)

Zhejiang University, Zhejiang University NGICS Platform, Hangzhou, China
{yfxia,gnip,zhangxuhong,liupeiyu,sji,zdzzlab}@zju.edu.cn

Abstract. The emergence of WebAssembly allows attackers to hide the malicious functionalities of JavaScript malware in cross-language interoperations, termed JavaScript-WebAssembly multilingual malware (JWMM). However, existing anti-virus solutions based on static program analysis are still limited to monolingual code. As a result, their detection effectiveness decreases significantly against JWMM. The detection of JWMM is challenging due to the complex interoperations and semantic diversity between JavaScript and WebAssembly. To bridge this gap, we present JWBinder, the first technique aimed at enhancing the static detection of JWMM. JWBinder performs a language-specific data-flow analysis to capture the cross-language interoperations and then characterizes the functionalities of JWMM through a unified high-level structure called Inter-language Program Dependency Graph. The extensive evaluation on one of the most representative real-world anti-virus platforms, VirusTotal, shows that JWBinder effectively enhances anti-virus systems from various vendors and increases the overall successful detection rate against JWMM from 49.1% to 86.2%. Additionally, we assess the side effects and runtime overhead of JWBinder, corroborating its practical viability in real-world applications.

Keywords: Malware and Unwanted Software · Software Security · Web Security

1 Introduction

JavaScript is a highly prevalent scripting language known for its significant role in web application development [3]. In recent years, it has also extended its influence beyond the browser with the support of NodeJS [19]. The ubiquity of JavaScript naturally makes it a target for attackers, giving rise to a variety of attack vectors, such as CryptoJacking [31], Drive-by-download attacks [34] and JavaScript Skimmers [2]. Additionally, attackers have now started exploiting Open Source Software (OSS) by injecting malicious JavaScript third-party packages into public registries like NPM [1].

G. Tsudik et al. (Eds.): ESORICS 2023, LNCS 14345, pp. 255–276, 2024.
https://doi.org/10.1007/978-3-031-51476-0_13

To counter these threats, current anti-virus solutions employ sophisticated program analysis techniques for malicious JavaScript detection [6,7,20,25,26,28–30,36,41–43]. Such approaches can be partitioned into two categories: static and dynamic approaches. Static approaches extract code features of varying granularities (e.g., Abstraction Syntax Tree (AST) [26] and Program Dependence Graph (PDG) [28]) from JavaScript without executing it. These features are then used for machine learning techniques [26,28,36] or program similarity analysis [41,43] to differentiate benign and malicious code. On the other hand, dynamic approaches detect abnormal JavaScript behavior (e.g., sensitive API calls) by running it in a honey client or sandbox [6,20]. Each approach has its own strengths and weaknesses. However, dynamic approaches are often burdened with considerable runtime overhead and struggle to detect malicious behaviors only manifesting under specific configurations. Thus, static approaches often form the preferred choice for anti-virus solutions due to their scalability and efficiency, consequently making them a prime target for attackers [27,37,40].

Despite the considerable ability of static approaches to detect malware, existing defense methods tend to assume that programs in the JavaScript ecosystem (e.g., Web and NodeJS) are composed purely of JavaScript. However, this assumption may no longer be held with the introduction of WebAssembly [22] in 2015. WebAssembly is an emerging binary code language that complements JavaScript. Initially designed for computation-intensive tasks, WebAssembly can be called upon by JavaScript programs through foreign language interfaces, which also provides new opportunities for attackers to create JavaScript-WebAssembly Multilingual Malware (JWMM) [24,37]. Specifically, attackers can conceal malicious behaviors within the interoperations between JavaScript and WebAssembly. Consequently, prior works that statically extract program features solely from JavaScript [25,26,28–30,36,41–43] could hardly identify these concealed malicious behaviors. Even the existing works [38] that consider malicious WebAssembly struggle to mitigate this threat because the malicious behaviors of JWMM are concealed behind the cross-language interoperations, which are unlikely to be identified with detectors focusing on a single language.

The effectiveness of JWMM against static approaches raises legitimate concerns. However, the detection of JWMM presents two major challenges:

C1: *Interoperation Complexity.* A fundamental step in characterizing JWMM involves understanding how JavaScript interacts with WebAssembly units in JWMM. This is far from a trivial task due to the complexity stemming from the intricate mechanisms upon which the interoperations of JavaScript and WebAssembly depend. For example, there are various interfaces to initialize a WebAssembly instance in JavaScript. Additionally, both languages can import and invoke functions from each other in adherence to the language standard [16]. Without recognizing these interoperations, it is difficult to unify the semantics of different language units in JWMM.

C2: *Semantics Diversity.* Existing works capture various patterns and features in sole language for distinguishing monolingual malware [25,29,38,43]. However, JavaScript and WebAssembly have disparate language semantics. Therefore, the

characterization of the JWMM's functionalities necessitates the consideration of both JavaScript and WebAssembly semantics. Furthermore, even if we consider the semantics of both JavaScript and WebAssembly, the definition of malicious patterns/features in multilingual programs remains an open issue.

In this paper, we present JWBinder, the first technique that characterizes the functionalities of JavaScript-WebAssembly multilingual programs to enhance the static detection of JWMM. To tackle C1, JWBinder presents a language-specific data-flow analysis to capture the interoperations between JavaScript and WebAssembly. Specifically, for the target JWMM, JWBinder constructs and traverses the PDG of its JavaScript unit. Then, by tracing the data dependencies flowing into and out of the foreign language interfaces, JWBinder can identify the concealed WebAssembly units in JWMM and detect how JavaScript and WebAssembly interact with each other. For example, the instantiation of WebAssembly which passes JavaScript external functions to WebAssembly instances, or the invocation point of a WebAssembly internal function in the JavaScript unit.

The solution of C2 relies on the crucial observation that, while WebAssembly's semantics differs significantly from JavaScript, it shares some unified features derived from JavaScript, such as similar control-flow instructions and basic data instructions [23]. Furthermore, WebAssembly can only invoke privileged system functions by importing them from JavaScript rather than through customized implementation. These homogeneous features allow us to design a uniform abstract representation that characterizes the functionalities of JavaScript-WebAssembly multilingual programs at a high level.

Based on the above insight, we propose a novel technique called *Static Semantics Reconstruction* (SSR). Leveraging the homogeneous features between JavaScript and WebAssembly, we first design a set of abstraction rules which encapsulate the semantics of WebAssembly at a high level. Following our abstraction rules, SSR generates a JavaScript-like abstract representation of the JWMM's WebAssembly units. Then, SSR integrates the abstract representations of WebAssembly units into JavaScript PDG to construct a uniform structure termed Inter-language PDG (IPDG), which characterizes the semantics within and across the JavaScript and WebAssembly units. Rather than designing ad-hoc heuristics to enumerate the malicious patterns of JWMM using the IPDG, SSR's final phase involves translating the IPDG and reconstructing a pure JavaScript program, which serves to elucidate the functionalities of the initial JWMM. The rationale is that since the IPDG is originally constructed following JavaScript abstract syntax, it can be naturally translated back to JavaScript by node traversing. Anti-virus solutions can further examine this reconstructed pure JavaScript program to determine the malignancy of the original JWMM.

The reconstruction of pure JavaScript programs offers two primary benefits. Firstly, it transforms the challenge of identifying multilingual malware into the more straightforward task of detecting monolingual malware. This allows the reuse of detection patterns/features designed for monolingual malware, thereby effectively addressing the semantic diversity problem. Secondly, with JWBinder

acting as a preliminary process, existing anti-virus solutions that concentrate solely on JavaScript can be employed directly to detect JWMM without any modifications, enhancing this approach's practicality for real-world applications.

To validate our design, we build a JWMM dataset based on 44,369 real-world JavaScript malware and evaluate JWBinder using this dataset. We have the following results. First, the approach is effective in enhancing the detection capabilities of well-known commercial Anti-Virus Systems (AV-Systems): With the application of JWBinder, the overall successful detection rate of VirusTotal [21] increases from 49.1% to 86.2%. Meanwhile, the number of AV-Systems successfully detecting malicious samples has increased from 4.1 to 8.3 on average. Second, JWBinder introduces nearly no side effects to benign programs: Processing a benign JavaScript-WebAssembly multilingual programs dataset and uploading them to VirusTotal, the results show minimal differences (0.5% false positive rate) compared to the original benign cases. This demonstrates that JWBinder does not induce suspicious behaviors which influence the detection of benign multilingual programs. Third, we investigate the generalization ability of JWBinder on AV-Systems from different vendors. The results show that more than 10 different AV-Systems significantly benefit from JWBinder. In particular, AV-Systems from different vendors may favor particular variants of JWBinder. Lastly, we find that our tool only requires, on average, 25.6 s to process a single JWMM program (with an average size of 282 KB), which is acceptable in comparison to previous JavaScript program analysis works [29,32]. Collectively, these results indicate that JWBinder is a practical tool for the enhancement of real-world anti-virus solutions.

In summary, this paper offers the following contributions:

- We propose the first program analysis technique designed to counter JWMM, which solutes the challenge of interoperation complexity and semantics diversity for analyzing JWMM.
- We implement the prototype of JWBinder, with a data-flow analysis framework for capturing cross-language interoperations in JWMM and a novel method termed static semantic reconstruction to characterize the unified functionalities of JWMM.
- We conduct a comprehensive evaluation demonstrating that our approach effectively enhances state-of-the-art anti-virus solutions and provides an analysis of their internal mechanisms.

To foster further research, we will release our experiment data and implementation at [14]. We believe that JWBinder provides valuable insights to detect JavaScript-WebAssembly multilingual malware.

2 Background and Motivation

2.1 WebAssembly

WebAssembly [22], a low-level binary instruction format, has become a fundamental web standard due to its secure and efficient characteristics. It facilitates

web development by enhancing the performance of applications and enabling seamless integration with JavaScript.

The WebAssembly binary includes multiple sections. Most notably, its code section holds the functional components, while the memory and data section manages linear memory for runtime behavior. The instructions of WebAssembly work at a low level to comprise simple operations such as arithmetic, control flow, and memory access. Some of the instructions share similar semantics to high-level languages (e.g., loop, branch, variable declaration/usage instructions), while others have special functionality corresponding to specific features (e.g., memory instructions, bit-level numeric instructions) of WebAssembly.

WebAssembly does not have a standard library, which means it can not directly access system APIs. To perform such functionalities, WebAssembly must import external functions from its host language (e.g., JavaScript). To achieve this, JavaScript uses foreign language interfaces (FFI) [16] to modularize and instantiate (that is, fulfill the imports of) WebAssembly modules and call exported WebAssembly functions.

2.2 JavaScript-WebAssembly Multilingual Malware

We define JavaScript-WebAssembly Multilingual Malware (JWMM) as a family of malware which hides malicious behaviors across the interoperations between JavaScript and WebAssembly. To evade the detection of anti-virus solutions, known JWMM often abuses WebAssembly binary for hiding sensitive instructions and data, and changing the control flow of JavaScript [24,37].

Alan et al. [37] developed Wobfuscator to generate JWMM automatically and demonstrated its efficacy against academic machine-learning-based classifiers. In this paper, we extend this investigation by testing JWMM's evasion abilities against a leading anti-virus platform, VirusTotal. We found that over half of the JWMM samples successfully evaded VirusTotal's detection, more details will be discussed in Sect. 4.

2.3 A Motivating Example

Figure 1 depicts a motivating example to illustrate how JWMM evades the detection of existing anti-virus solutions for JavaScript malware.

Figure 1.a shows a real-world pure JavaScript malware successfully detected by McAfee-GW-Edition [15]. To conduct the attack, it iteratively executes the **document.write** function (lines 7–9) to insert pre-defined malicious payloads (lines 2–5) into HTML. Figure 1.b and Fig. 1.c present the JavaScript part and WebAssembly part (in human-readable format) of the JWMM which perform equivalent malicious functionalities to Fig. 1.a.

Next, we explain how this JWMM evades detection by abusing the interoperations between JavaScript and WebAssembly. In Fig. 1.b, the JavaScript part of JWMM only needs to instantiate a WebAssembly instance by calling the FFI sequence (lines 2–6), and then invokes the internal function **foo** exported from

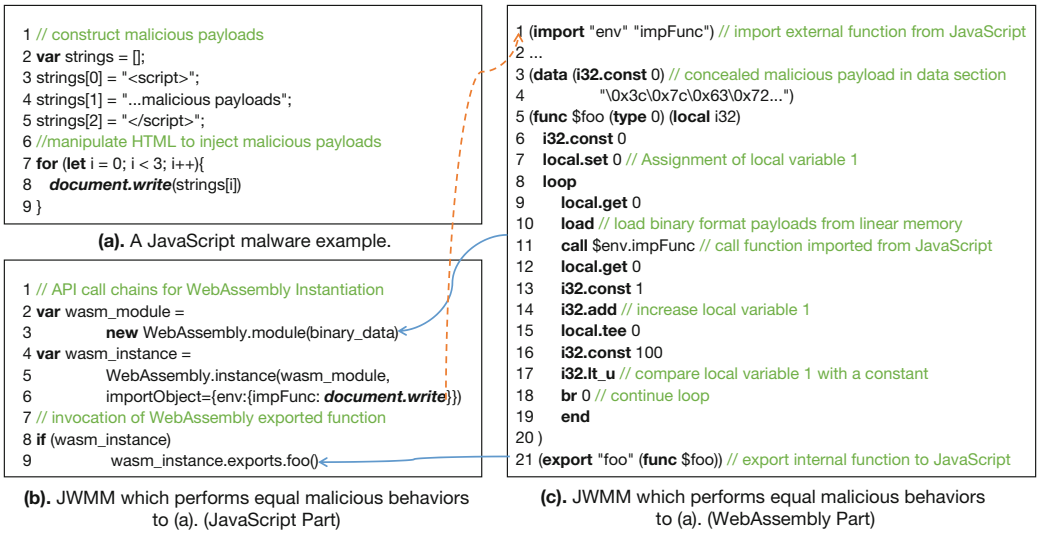

Fig. 1. An example of JWMM (b and c) and its equivalent JavaScript malware (a) (Color figure online)

the WebAssembly part (the blue solid line). In other words, it conceals the malicious functionalities in the WebAssembly. While in Fig. 1.c, the WebAssembly part imports the **document.write** from JavaScript (the red dashed line) and iteratively executes the function in a loop (lines 8–19) with transformed binary-format malicious payloads stored in its data section (lines 3–4).

Existing anti-virus solutions designed for pure JavaScript are not effective in detecting such JWMM. Also, merely analyzing the WebAssembly binary additionally could hardly reveal cross-language malicious functionalities if we are not aware of the exact imported function from JavaScript (i.e. **document.write**).

This example not only illustrates how JWMM evades the detection of monolingual anti-virus solutions, thereby emphasizing the necessity for a cross-language, comprehensive analysis, but also motivates our insights to reconstruct JWMM to a monolingual format, as depicted in Fig. 1.a. By doing so, we can expose its malicious functionalities for monolingual anti-virus solutions to identify.

3 JWBinder

This section describes our technique approach. We start with an overview (Sect. 3.1) of JWBinder and then elaborate two critical phases: language-specific data-flow analysis (Sect. 3.2) and static semantic reconstruction (Sect. 3.3).

3.1 Approach Overview

Figure 2 gives the overview of JWBinder. The input of JWBinder is the JavaScript-WebAssembly multilingual program under detection. With the input, JWBinder works in two phases. In Phase 1, JWBinder first abstracts the

JavaScript unit of the multilingual program to construct the JavaScript PDG. We adopt the definition of PDG followed Fass et al. [29], which integrates the control-flow and data-flow dependencies into the Abstract Syntax Tree of the JavaScript unit. Next, JWBinder traverses the PDG to perform a bi-directional data-flow analysis which captures the interoperation between JavaScript and WebAssembly.

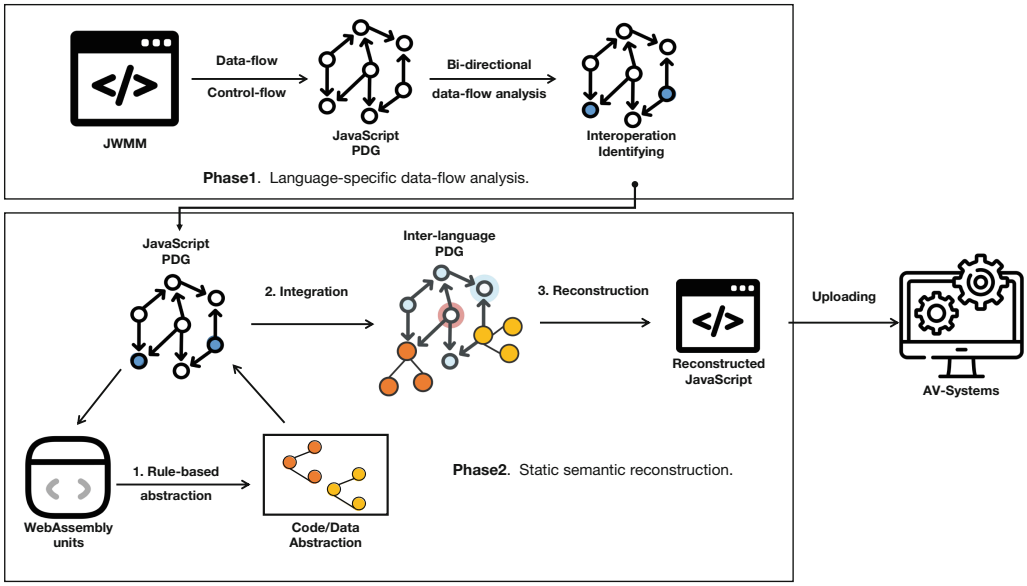

Fig. 2. An overview of JWBinder

After Phase 1, the functionalities of WebAssembly are still invisible on the JavaScript PDG. Thus, JWBinder starts its Phase 2 to construct a language-agnostic structure for characterizing the functionalities of multilingual programs. In Phase 2, JWBinder first extracts the WebAssembly units in the multilingual program based on the identified interoperations in Phase 1. For every individual WebAssembly unit, JWBinder abstracts its code and data sections following a set of abstraction rules, which concentrate on the homogeneous semantics between JavaScript and WebAssembly. The abstractions of WebAssembly units are then integrated into the JavaScript PDG to construct a uniform language-agnostic structure termed Inter-language Program Dependency Graph (IPDG). Finally, JWBinder reconstructs a pure JavaScript program based on the IPDG, transforming the problem of detecting multilingual malware back into monolingual malware. As a result, JWBinder outputs pure JavaScript programs that statically characterize the original multilingual program's functionalities, making these outputs ready for detection by monolingual anti-virus solutions (e.g., AV-Systems).

3.2 Phase 1: Language-Specific Data-Flow Analysis

At a high level, a JavaScript-WebAssembly multilingual program comprises a JavaScript program that interacts with WebAssembly through specific interfaces, with the WebAssembly units concealed in the JavaScript unit. To characterize the functionalities of the multilingual program, JWBinder should be able to recognize how JavaScript interacts with WebAssembly units. To this goal, the first phase works in two major steps as elaborated below.

PDG Generation. JWBinder first abstracts the JavaScript unit of the multilingual program for an in-depth data-flow analysis. Given the JavaScript unit, JWBinder generates its PDG following the definition of Fass et al. [29]. The PDG is built on the JavaScript AST, incorporating control-flow and data-flow dependencies during AST traversal. In the PDG, The control-flow dependencies present a decision pathway, delineating whether a specific execution path would be pursued. Concurrently, the data-flow dependencies offer insights into the interrelationships among different variables within the program.

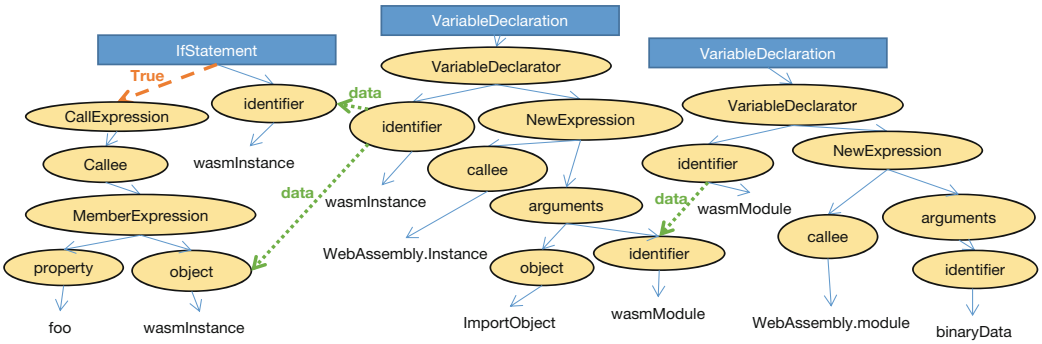

Fig. 3. Example PDG of Fig. 1.b (some nodes are simplified for clarity) (Color figure online)

For instance, Fig. 3 is the PDG of the JavaScript program shown in Fig. 1.b. The control dependency labeled as "True" (red dashed line) suggests that the program path will be pursued only when the preceding condition holds true. Meanwhile, the data dependencies (green dot lines) present the definition and usage relationships of different variables (e.g., wasmInstance and wasmModule). Both these dependencies are valuable for identifying the interoperation between JavaScript and WebAssembly.

Interoperation Identification. Once the JavaScript PDG is constructed, JWBinder conducts a bi-directional data-flow analysis to capture the interoperation between JavaScript and WebAssembly. The multilingual interoperation patterns normally contain the modularization/instantiation of WebAssembly and the invocation of WebAssembly properties. The former process enables JavaScript to pass external properties into WebAssembly, while the latter makes

JavaScript invoke the internal properties of WebAssembly. For example, in Fig. 1.b, the JWMM compiles a WebAssembly module (*wasmModule* in lines 2–3), employs the module to instantiate a WebAssembly instance (*wasmInstance* in lines 4–6) with external function *document.write()*, and finally calls an internal function from the WebAssembly instance (*wasmInstance.foo()* in line 9).

To capture the above interoperation patterns, JWBinder traverses the PDG to identify the key APIs and properties (listed in Fig. 4 within the Appendix) for WebAssembly modularization or instantiation. Upon encountering one of these key APIs, JWBinder marks the relevant PDG nodes as interoperation positions and discerns the external properties passed into WebAssembly through the APIs. Then JWBinder undertakes a forward data-flow analysis to trace the locations where JavaScript invokes internal properties of the WebAssembly instances. Meanwhile, a backward data-flow analysis is also conducted to trace the concealed binary format WebAssembly for further abstractions in Phase 2. We detail the algorithm in Algorithm 1 within the Appendix. Ultimately, JWBinder generates a JavaScript PDG where the cross-language interoperations are distinctly marked, providing a clear depiction of the intricate connections between JavaScript and WebAssembly.

3.3 Phase 2: Static Semantic Reconstruction

JWBinder performs the static semantic reconstruction (SSR) to characterize the functionalities of JWMM after the interoperations between JavaScript and WebAssembly have been identified. The key insight behind SSR is the integration of JavaScript and WebAssembly semantics into a uniform representation based on their homogeneous features. In particular, Phase 2 works in three steps as elaborated below.

Rule-Based Abstraction. A joint analysis for multilingual programs requires a uniform representation that merges both JavaScript and WebAssembly semantics. Since the main module of JWMM is the JavaScript unit, JWBinder integrates the semantics of WebAssembly units into the JavaScript PDG generated in Phase 1 for multilingual analysis. However, this is a non-trivial process due to the disparate semantics of JavaScript and WebAssembly. Most importantly, JavaScript is a dynamically typed language, while WebAssembly is a statically typed language, which has diverse data types. The disparity makes it difficult to unify the cross-language semantics within JWMM.

JWBinder addresses this challenge by introducing a set of abstraction rules, which extracts high-level semantics from WebAssembly to unify different language units. The critical insight of our abstraction rules is that they focus on homogeneous features between JavaScript and WebAssembly, such as similar data-related operations (e.g., variable declaration and usage) and logic-related instructions (e.g., loop structures and condition statements). Based on this, JWBinder is able to translate the semantics of WebAssembly to JavaScript-like AST nodes and bridge the semantic gap between JavaScript and WebAssembly.

In general, the abstraction rules fall into two categories: code abstraction and data abstraction, targeting code and data sections mentioned in Sect. 2. We now introduce different categories of abstract rules.

The **code** abstraction rules aim to characterize the functionalities of internal functions defined in WebAssembly's code section. For example, in Fig. 1.b, the definition of the function *foo* (line 9) lies in WebAssembly's code section. SSR reveals its functionality by abstracting the corresponding WebAssembly instructions (lines 5–20 in Fig. 1.c). WebAssembly code instructions execute on a stack machine, in that instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing or returning (pushing) result values [23]. To abstract these instructions, we should first capture the data relationships between stack values. Therefore, we model WebAssembly instructions based on their effects on the stack, following the simulation of [38]. According to the stack state, we design specific rules for different code instructions to extract their high-level semantics and abstract them to JavaScript ES6 [13] syntax units. We choose ES6 syntax abstractions because it helps to characterize WebAssembly instructions within JavaScript syntax, which strengthens the homogeneity between these two languages.

Table 1. Abstraction rules

Instruction	Stack Operation	Abstractions	Equivalent JavaScript
get_local v	push(v)		
i32.const c	push(c)	VariableDeclaration -> [id -> C_n , init -> c, kind -> const]	const C_n = c;
set_local v	pop() -> e	AssignStatement -> [id -> v , init -> e]	v = e ;
i32.mul	pop() -> e_1, pop() -> e_2, push(e_1 * e_2)	VariableDeclaration -> [id -> V_n , kind -> let, init -> BinaryExpression -> [left -> e_1, right -> e_2, op: *]]	let V_n = e_1 * e_2;
i32.popcnt	pop() -> e, push(popcnt())	VariableDeclaration -> [id -> V_n , kind -> let, init -> CallExpression -> [callee -> popcnt, arguments -> [e]]	let V_n = popcnt(e);
loop/block g		LabelStatement -> [body -> ForStatement -> [init, test->true, update, body -> [...]] , label -> g]	g: for(;true;){...};
if I	pop() -> e	IfStatement -> [test -> e, body-> [...], alternate -> [...]]	if (e) {...};
br g		BreakStatement -> [label -> g] (in block context) ; ContinueStatement -> [label -> g] (in loop context)	break g ; (in block context) continue g ; (in loop context))
call f	paraNum(f) -> n, for(i=0;i<n;i++) pop -> e_i	CallExpression -> [callee - > f, arguments -> [e_1, e_2, ... , e_n]]	f(e_1,e_2,...,e_n);
call_indirect	pop() -> e, paraNum(f) -> n, for(i=0;i<n;i++) pop -> e_i	CallExpression -> [callee - > f_e, arguments -> [e_1, e_2, ... , e_n]]	f_e(e_1,e_2,...,e_n);
data (type) "payloads"		VariableDeclaration -> [id -> memory , kind -> var, init -> BinaryExpression -> [left -> e_1, right -> e_2, op: *]]	var memory = "payloads";

Table 1 shows a subset of the abstraction rules. Due to the space limit, we present the full version on our website [14] and only list abstraction rules for the key data-flow and control-flow instructions. For instructions that do not assign variables or manipulate control flows, such as "local_get" which only fetches an existing local variable and pushes it onto the stack, we models their stack operation without generating abstractions.

Data-Flow Instruction. WebAssembly functions operate their variables following the "def-use" chain like high-level programming languages. We abstract such data-flow instructions as the foundation to characterize the semantics of WebAssembly's code section. Specifically, we abstract instructions that manipulate local/global variables to "AssignStatement" syntax units. As shown in row 4, Table 1, *"set_local v"* consumes a stack value e and assigns e to local variable v. The abstraction of *"set_local v"* indicates this instruction has equivalent functionality compared to JavaScript code *"$v = e$"*. Instructions that push new values to the stack are abstracted to "VariableDeclaration" syntax units: *"i32.const c"* declares a new constant and pushes it onto the stack. Its abstraction is equivalent to *"const C_n = c;"*, where *"C_n"* is a randomly generated name for evading name conflicts. Similarly, the instruction *"i32.mul"* and *"i32.popcnt"* are abstracted to new variable declarations. In particular, operators that are unsupported by JavaScript are simulated to user-defined functions (e.g., *popcnt* in row 5). The rest of Table 1 shows code abstraction rules for WebAssembly control flow instructions.

Control-Flow Instruction. WebAssembly supports control flow instructions, which are similar to high-level languages such as JavaScript. These instructions are significant for constructing a uniform representation. As shown in Table 1, the instructions *"loop"* and *"block"* are both abstracted to labeled "ForStatement" syntax units in Table 1. However, they construct different contexts where the *"br"* instruction is abstracted to different syntax units (i.e., Break and Continue), resulting in diverse control flow structures. *"if I"* is abstracted to "IfStatement", which is a homogeneous structure in JavaScript. In particular, the *"call"* and *"call_indirect"* instructions are non-deterministic because they can either call WebAssembly internal functions or call external functions imported from JavaScript. We determine the exact function referring to the interoperations identified in Phase 1.

We now describe our **data** abstraction rules. The data abstraction rules characterize suspicious values hidden in the WebAssembly linear memory. For example, in Fig. 1.c, the malicious payload is stored in the linear memory and loaded when the function *foo* executes. WebAssembly linear memory is related to two key sections: the memory section and the data section, where the former configures the linear memory and the latter initializes it. Besides static initialization, the linear memory can also be manipulated at runtime through specific instructions such as *store*. However, capturing the exact memory through simulation is unaffordable because precise simulation of all the memory manipulation consumes high overhead. Thus, we limit the scope of data abstraction to the initialization of linear memory. To be specific, we analyze all of the segments in WebAssembly data sections and abstract them to "VariableDeclaration" syntax units as shown in the last row of Table 1, which means every individual data segment is declared as a unique global variable.

Finally, both the code and data abstractions are presented as JavaScript-like AST nodes, facilitating seamless integration into the JavaScript PDG.

IPDG Integration and JavaScript Reconstruction. Once JWBinder finishes abstracting identified WebAssembly units. The next step is to integrate the WebAssembly abstractions into the JavaScript PDG so that we can obtain an inter-language PDG. The integration takes two kinds of inputs: interoperation positions identified in Phase 1 and the abstraction results generated in Phase 2. Specifically, JWBinder replaces the PDG nodes identified as invocation positions of WebAssembly functions with their code abstraction, thereby unveiling the previously hidden semantics. Concurrently, JWBinder integrates the data abstraction of WebAssembly units into their instantiation positions, signifying the initialization of their linear memories.

At present, the IPDG is directly reconstructed into a pure JavaScript program using Escodegen [8], making it compatible with existing monolingual anti-virus systems. Escodegen, which is a well-tested tool widely adopted in previous work [27,37,39], can generate JavaScript programs from abstract representations following ES6 standards. Nevertheless, we believe that the IPDG structure could potentially serve as a foundation for developing advanced multilingual analysis techniques. We aim to explore this avenue in our future work.

4 Evaluation

Using our JWBinder implementation in Sect. 3, the evaluation of our approach is guided by four research questions below:

RQ1: How effective is JWBinder when deployed with real-world anti-virus solutions?

RQ2: Does JWBinder introduces any side effects to the benign JavaScript-WebAssembly multilingual programs?

RQ3: What is the generalization ability of JWBinder on different commercial anti-virus solutions?

RQ4: How efficient is JWBinder in terms of its runtime overhead.

4.1 Experiment Setup and Preliminary Study

We first describe our experiment settings and conduct a preliminary study to illustrate the weakness of existing works against JWMM.

1) JWMM Dataset. To evaluate the effectiveness and efficiency of JWBinder, a multilingual malware dataset is required. However, to our best knowledge, such a dataset is not available. Meanwhile, it may not be feasible to curate the ground truth for large complex real-world programs. Thus, we take inspiration from prior works [33,37] to construct a JWMM dataset (JWBench) deriving from real-world JavaScript malware. Below, we detail our dataset construction:

- Step-1: **Initial malware selection and filtering**. The initial malware dataset consists of 44369 samples, with 39,450 samples from the Hynek Petrak JavaScript malware collection [11], 3562 samples from VirusTotal [21], and 1357 samples from the GeeksOnSecurity [10]. To ensure the semantic validity of the malware samples, We use Esprima [9], a popular standard-compliant JavaScript parser, to vet the initial samples. Additionally, we filter the samples which rely on *cc_on* statements to perform malicious behaviors. *Cc_on* statement is a special mechanism that only works in IE browser [18], which generates executable comments. Thus the detection of such samples is beyond our research scope. The final JavaScript malware dataset consists of 21191 JavaScript samples, termed JWBench$_o$. We believe it is of more validity for reasonable evaluation.
- Step-2: **JWMM generation**. We employed the technique proposed by Alan et al. in [37] to convert real-world JavaScript malware to JWMM for evasion detection. We implement Wobfuscator and apply it to malicious samples from step 1, following the original experiment configuration in [37]. As a result, the final JWBench contains 21191 JWMM samples.

2) Baseline Anti-virus Solutions.

To perform a large-scale, representative experiment, we leverage VirusTotal as our baseline anti-virus solution. VirusTotal is an online platform housing 74 real-world commercial AV-Systems, including renowned ones like McAfee [15], Microsoft [17], and BitDefender [5], which makes it the chief target for comprehensive evaluations. Currently, there are 59 AV-Systems providing detection services for malicious JavaScript. When provided a JavaScript program as input, VirusTotal generates a detection report containing binary results (malicious or benign) from these AV-Systems.

We employed two overall metrics to assess the performance of the AV-Systems on VirusTotal:

Successful Detection Rate. A sample is considered successfully detected if no less than a certain threshold number of AV-Systems on VirusTotal accurately identify it as malicious. We set this threshold at two to avoid false positives detection referring to [44]. The Successful Detection Rate (**SDR**) signifies the ratio of successfully detected samples to the overall samples.

Average Detected Engines. From a defensive perspective, it is desirable for as many AV-Systems as possible to successfully detect a malicious sample. To this end, we introduce the metric Average Detected Engines (**ADE**) as another metric to evaluate the capability of AV-Systems on VirusTotal in handling JWMM. The ADE represents the mean number of AV-Systems that successfully detect each malicious sample.

3) Performance of Existing Anti-virus Solutions. At a high level, JWMM consists of JavaScript and WebAssembly units. However, existing anti-virus solutions might address JWMM from a monolingual perspective without completely considering the interoperations between JavaScript and WebAssembly.

This oversight substantially undermines the detection capability of these anti-virus solutions when dealing with JWMM.

Table 2. The detection result of VirusTotal on different datasets

	JWBench_o	JWBench
Successful Detection Rate	99.9%	47.6%
Average Detected Engines	22.4	3.3

In Table 2, we list the comparison of VirusTotal's detection results on Different Datasets. As seen in Table 2, on JWBench_o, the SDR and ADE of VirusTotal is 99.9% and 22.4, showcasing its effectiveness against pure JavaScript malware. However, these metrics drop to 47.6% and 3.3 on JWBench, meaning that more than half of the JWMM samples can elude detection by VirusTotal. This result underscores the real-world security threats posed by JWMM.

We further investigate whether anti-virus solutions that target the malicious WebAssembly can detect JWMM effectively. Specifically, we first utilize JWBinder's data-flow analysis component to extract 10,000 WebAssembly binaries from JWBench. Subsequently, we employ MinerRay [38], a proven static detector for malicious WebAssembly (e.g., CryptoMiners), to scrutinize these extracted binaries. However, MinerRay fails to detect any suspicious activities in all the samples. This result underscores the notion that even specialized anti-virus solutions, despite their deep semantic understanding of WebAssembly, struggle to detect JWMM. This is predominantly because their focus remains limited to heuristic monolingual features.

4.2 Effectiveness of JWBinder (RQ1)

The effectiveness of the JWBinder is evaluated through a comparison of the AV-Systems' performance on the original JWMM input programs against their performance after JWBinder has been applied. For this evaluation, we randomly select 10,000 samples from the JWBench and measure the system's performance using the metrics outlined in 4.1. The primary objectives of JWBinder are to enhance SDR and the ADE of VirusTotal against the selected JWMM samples.

Table 3 shows the successful detection rate and average detected engines of VirusTotal with/without the application of JWBinder. The first column gives the detection results of VirusTotal without JWBinder, which serves as a baseline. The rest of the column shows the results after applying JWBinder with different levels of abstraction in semantic reconstruction.

JWBinder_c refers for JWBinder which merely applies code abstraction in SSR and JWBinder_d refers for JWBinder which merely applies data abstraction in SSR. The fourth column, JWBinder_a, corresponds to the complete version of JWBinder which combines the results of individual JWBinder_c and JWBinder_d. The values in brackets indicate the difference from the baseline. For example,

the second column indicates that JWBinder$_c$ can successfully detect 16.1% more malicious samples compared to the baseline, while failing to detect 0.5% of the samples that the baseline can originally detect.

Table 3. The detection result of VirusTotal on 10k JWMM processed by JWBinder

	Baseline	JWBinder$_c$	JWBinder$_d$	**JWBinder$_a$**
SDR	49.1%	64.7% ($\frac{+16.1\%}{-0.5\%}$)	81.0% ($\frac{+33.5\%}{-1.6\%}$)	**86.2%**($\frac{+37.1\%}{-0.0\%}$)
ADE	4.1	5.0	7.5	**8.3**

Table 3 illustrates that all variants of JWBinder substantially improve the performance of AV-Systems on VirusTotal in terms of both SDR and ADE. Specially, JWBinder$_c$ and JWBinder$_d$ achieve 15.6%/31.9% increment for SDR and are successful in having each malicious sample detected by 0.9/3.4 additional engines, respectively. Moreover, the complete JWBinder$_a$ attains an SDR of 86.2% and an ADE of 8.3.

Of all the malicious samples, either JWBinder$_c$ or JWBinder$_d$ successfully identifies 59.5% of them. There are also 5.2% and 21.5% unique samples detectable exclusively by JWBinder$_c$ and JWBinder$_d$, respectively. This outcome serves as an ablation study, demonstrating that different SSR levels contribute to the detection of various samples.

The SSR applied by JWBinder could disrupt detection results if it provides AV-Systems with information that may not favor their detection capabilities. For instance, signature-based detection does not respond significantly to code-level information. We attribute the slight decrease in SDR for JWBinder$_c$ (-0.5%) and JWBinder$_d$ (-1.6%) to this factor. However, by merging the results from both individual SSR applications, JWBinder can enhance detection without any negative impact (i.e., 0% decrement).

4.3 Side Effects (RQ2)

While JWBinder has demonstrated significant improvements in JWMM detection, it is essential to evaluate its potential side effects. Specifically, we need to ensure that JWBinder enhances the effectiveness of AV-Systems on VirusTotal without introducing suspicious behaviors to the benign samples under detection.

A false positive detection occurs when the benign program is wrongly identified as malware by more than one engine. For RQ2, we evaluate the number of false positive results of VirusTotal when runs on benign samples processed by JWBinder. Following the experiment settings in Sect. 4.1, we generate a benign JavaScript-WebAssembly multilingual programs dataset from JS150k [4], which contains 150000 JavaScript source files. Due to the scaling issues, we randomly select 1000 of them which VirusTotal deems benign and manually confirm the detection results. To evaluate whether JWBinder introduces side effects on the

samples it processes, we compare the AV-Systems' false positive detections on the original benign input programs when applying JWBinder.

Our experimental results reveal a relatively low false positive rate introduced by JWBinder. Of the 1000 benign samples, JWBinder$_c$ and JWBinder$_d$ only cause 4 and 1 false positive detections, respectively. As a result, the JWBinder$_a$ registers a false positive rate of only 0.5%. Given this relatively low false positive rate compared to the considerable enhancements in JWMM detection, we conclude that JWBinder introduces minimal side effects when processing JavaScript-WebAssembly multilingual programs.

4.4 Generalization Ability of JWBinder (RQ3)

In this research question, we evaluate the generalization ability of JWBinder across different commercial AV-Systems on VirusTotal. Furthermore, we aim to deduce the internal mechanisms used in these AV-Systems based on the distribution of results by comparing their benefits from different variants of JWBinder.

Table 4 shows the detection results of different commercial AV-Systems. Due to the limitation of length, we only list 5 representative AV-Systems on VirusTotal (the complete table is deferred to Table 5 in the Appendix). The results show that different levels of SSR benefit certain detectors more rather than others. For example, JWBinder$_c$ and JWBinder$_d$ both help Google and Cyren achieve 20%+ SDR increment, from which we can deduce that these two AV-Systems are likely to classify malicious JavaScript according to either code-level features and data-level features. As a result, JWBinder$_a$ has increased the SDR of Google and Cyren from 31.1%/29.4% to 61.3%/55.9%.

Table 4. The successful detection rate of individual AV-System on JWMM processed by JWBinder, highest SDR in **BOLD**

AV-System	Baseline	JWBinder$_c$	JWBinder$_d$	JWBinder$_a$
Google	31.1%	53.3%	59.4%	**61.3%**
Cyren	29.4%	50.5%	53.4%	**55.9%**
McAfee-GW-Edition	0%	47.5%	2.4%	**47.7%**
BitDefender	14.2%	19.7%	39.5%	**39.7%**
Microsoft	19.3%	16.0%	35.0%	**45.7%**

Some AV-Systems favor particular features more than others. For example, with JWBinder$_c$ revealing the code-level features of JWMM, McAfee-GW-Edition's SDR has increased from 0% to 47.5%. However, the increment is merely 2.4% with JWBinder$_d$, which shows that McAfee-GW-Edition is relatively insensitive to data-level features. In Table 4, the majority of AV-Systems (4 of 5) gain significantly from JWBinder$_d$, which corresponds to previous research [35] that signature-based matching is wide-adopted in AV-Systems.

4.5 Efficiency of JWBinder (RQ4)

Lastly, we evaluate the run-time performance of JWBinder. Since JWBinder runs the analysis of each JWMM on a single core, the reported runtime corresponds to a single CPU. There are two most time-consuming steps of JWBinder: the data-flow analysis on the JavaScript side and the SSR process which parses and abstracts the WebAssembly binary. Usually, a JWMM file has a much more complex AST than a traditional JavaScript file, making it time-consuming to be analyzed either dynamically or statically. Alan et al. [37] shows that the execution time JWMM may increase at most 2079.21% with 363.52% larger size compared to traditional JavaScript programs. On average, JWBinder needs 10.0 s for data-flow analysis to capture the interoperations between JavaScript and WebAssembly. Besides, it needs on average 15.6 s for SSR. Specially, the corresponding median times are 0.7 s and 6.7 s, while the maximum amounts of time are 207.5 s and 202.2 s, predominantly when JWBinder processes exceptionally large JWMM files (>1 Mb), with the details provided in Fig. 5 within the Appendix. This result is competitive compared to previous works for large-scale malicious JavaScript detection [29,32].

In particular, we could complete the data-flow analysis for 87.8% of our JWMM set in less than 10 s and SSR for 83.5% of them in less than 20 s. This efficiency enables JWBinder to effectively augment existing AV-Systems for detecting JWMM from the wild.

5 Limitation

Our current implementation of JWBinder effectively enhances the performance of existing anti-virus solutions. However, it also has a few limitations:

- **Threat of Run-time Code/Data Generation**. JWBinder aims to enhance existing anti-virus solutions during the static analysis phase. Therefore, it is insensitive to run-time behaviors. For example, attackers can dynamically construct malicious payloads in WebAssembly to evade our static data abstractions or generate code at runtime which is beyond the scope of our reconstructed JavaScript. However, none of the existing static approaches can effectively solve this problem, which can only rely on hybrid approaches.
- **Obfuscation**. Existing JavaScript obfuscation techniques may break the semantics required for taint analysis. Currently, JWBinder is not equipped with a de-obfuscation component. However, this threat can be mitigated with on-the-shelf de-obfuscation tools such as [7,12].

6 Conclusion

In this paper, we propose JWBinder, the first technique for enhancing the detection of JavaScript-WebAssembly multilingual malware (JWMM). JWBinder captures the interoperations between JavaScript and WebAssembly and then

reconstructs a statically equal JavaScript program, which characterizes the hidden malicious behaviors of JWMM through a novel uniform structure called Inter-language Program Dependency Graph. Our evaluation shows the reconstruction process can effectively enhance real-world AV-Systems. We also show to what extent can JWBinder benefit Anti-Virus Systems from different vendors. Finally, we evaluate the efficiency of JWBinder and prove it can be scalable to large JWMM in the real world.

Acknowledgements. This work was partly supported by NSFC under No. U1936215 and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform). We sincerely appreciate the anonymous reviewers for their insightful comments.

A Appendix

This appendix contains some supplementary material.

In particular, Fig. 4 lists a series of JavaScript WebAssembly interfaces for cross-language interoperations, which we leverage in the data-flow analysis.

Figure 5 presents JWBinder run-time performance depending on the JWMM size.

Algorithm 1 details the process for identifying the cross-language interoperations on PDG.

Finally, Table 5 shows extensive detection results of different AV-Systems. We list the top 15 AV-Systems due to the space limit.

Function/Property Name	Description
WebAssembly.compile() WebAssembly.compileStreaming() WebAssembly.Module	The modularization of WebAssmebly binaries. Return a module for Instantiation.
WebAssembly.instantiate() WebAssembly.instantiateStreaming() WebAssembly.Instance	The Instantiation of WebAssmebly module. Return a instance which can call WebAssembly functions.
Instance.exports	Export WebAssembly internal properties to JavaScript.

Fig. 4. WebAssembly modularization/instantiation functions and properties

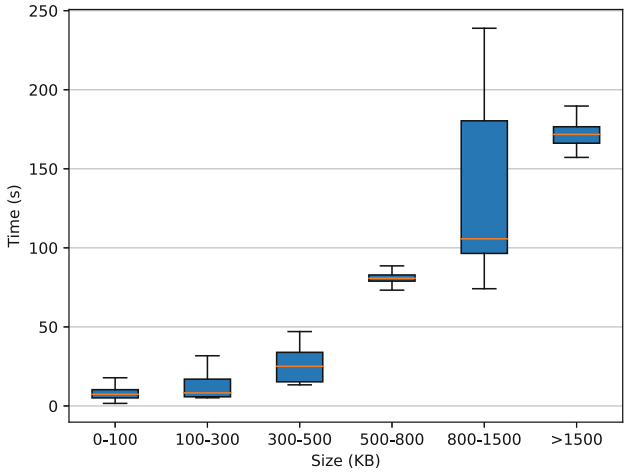

Fig. 5. Run-time performance of JWBinder depending on the JWMM size

Table 5. The successful detection rate of every individual AV-System on JWMM processed by JWBinder, highest SDR in **BOLD**

AV-System	Baseline	JWBinder$_c$	JWBinder$_d$	JWBinder$_a$
Google	31.1%	53.3%	59.4%	**61.3%**
Cyren	29.4%	50.5%	53.4%	**55.9%**
McAfee-GW-Edition	0%	47.5%	2.4%	**47.7%**
Microsoft	19.3%	16.0%	35.0%	**45.7%**
Rising	27.5%	2.6%	44.5%	**45.3%**
Arcabit	14.2%	20.0%	39.8%	**39.9%**
MicroWorld-eScan	14.2%	20.0%	39.8%	**40.0%**
FireEye	14.2%	19.9%	39.5%	**39.8%**
ALYac	13.8%	19.6%	38.9%	**39.4%**
GData	14.2%	19.0%	38.3%	**39.3%**
Emsisoft	14.0%	18.0%	36.9%	**38.6%**
BitDefender	14.2%	19.7%	39.5%	**39.7%**
VIPRE	14.1%	19.5%	39.1%	**39.5%**
MAX	14.2%	19.9%	39.6%	**39.8%**
Ikarus	2.07%	13.7%	23.6%	**24.0%**

Algorithm 1. Algorithm for identifying cross-language interoperations

```
Input: The Original PDG P
Output: The New PDG with Interoperation Marking P'
P' ⟵ P;
Function TraversePDG(P'):
    foreach child ∈ getChildren(P') do
        TraversePDG(child);                          ▷ Iteratively traversing the PDG
        if hasKeyAPI(P') then
            BackwardAnalysis(P');                    ▷ Start bi-directional data-flow analysis
            ForwardAnalysis(P');
        end
    end
End Function;
Function BackwardAnalysis(node):
    Worklist ⟵ [];
    Worklist.push(node);
    while Worklist ≠ NULL do
        CurNode = Worklist.pop()                     ▷ Backward tracing data-flow dependent parents
        foreach parent ∈ getDataParent(P', CurNode) do
            Worklist.push(parent)
            if isImportProperty(parent) then
                P' ⟵ markInteroperation(P', parent)
                                                     ▷ Mark interoperations when encountering imported items
            end
        end
    end
End Function;
Function ForwardAnalysis(node):
    Worklist ⟵ [];
    Worklist.push(node);
    while Worklist ≠ NULL do
        CurNode = Worklist.pop()                     ▷ Forward tracing data-flow dependent children
        foreach child ∈ getDataChildren(P', CurNode) do
            Worklist.push(child)
            if isExportProperty(child) then
                P' ⟵ markInteroperation(P', child)
                                                     ▷ Mark interoperations when encountering exported items
            end
        end
    end
End Function;
```

References

1. Npm (2021). https://www.npmjs.com/
2. The year in web threats: web skimmers take advantage of cloud hosting and more (2021). https://unit42.paloaltonetworks.com/web-threats-trends-web-skimmers/
3. Github top programming languages (2022). https://octoverse.github.com/2022/top-programming-languages
4. 150k Javascript dataset (2023). https://www.sri.inf.ethz.ch/js150
5. Bitdefender (2023). https://www.bitdefender.com/
6. Box.js, a sandbox to analyze malicious Javascript (2023). https://github.com/CapacitorSet/box-js
7. De4js, Javascript deobfuscator and unpacker (2023). https://lelinhtinh.github.io/de4js/
8. Escodegen (2023). https://github.com/estools/escodegen
9. Esprima (2023). https://esprima.org/
10. Geeks on security malicious Javascript dataset (2023). https://github.com/geeksonsecurity/js-malicious-dataset

11. Petrak, H.: Javascript malware collection (2023). https://github.com/HynekPetrak/javascript-malware-collection
12. Javascript deobfuscator (2023). https://deobfuscate.io/
13. Javascript es6 standard (2023). https://www.w3schools.com/js/js_es6.asp
14. Jwbinder source code and data (2023). https://github.com/JWBinderRepository/JWBinder
15. Mcafee (2023). https://www.mcafee.com/
16. Mdn web docs (2023). https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface
17. Microsoft defender antivirus (2023). https://www.microsoft.com/
18. Microsoft, internet explorer (2023). https://www.microsoft.com/download/internet-explorer
19. Nodejs (2023). https://nodejs.org/
20. Sandbox for semi-automatic Javascript malware analysis (2023). https://github.com/HynekPetrak/malware-jail
21. Virustotal (2023). https://www.virustotal.com/
22. Webassembly (2023). https://webassembly.org/
23. Webassembly core specification (2023). https://www.w3.org/TR/2022/WD-wasm-core-2-20220419/syntax/instructions.html
24. Webassembly is abused by ecriminals to hide malware (2023). https://www.crowdstrike.com/blog/ecriminals-increasingly-use-webassembly-to-hide-malware/
25. Yara (2023). https://virustotal.github.io/yara/
26. Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C.: ZOZZLE: fast and precise in-browser Javascript malware detection. In: 20th USENIX Security Symposium (USENIX Security 11) (2011)
27. Fass, A., Backes, M., Stock, B.: Hidenoseek: camouflaging malicious Javascript in benign asts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1899–1913 (2019)
28. Fass, A., Backes, M., Stock, B.: Jstap: a static pre-filter for malicious Javascript detection, pp. 257–269 (2019)
29. Fass, A., Somé, D.F., Backes, M., Stock, B.: Doublex: statically detecting vulnerable data flows in browser extensions at scale. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, p. 1789–1804 (2021)
30. Kolbitsch, C., Livshits, B., Zorn, B.G., Seifert, C.: Rozzle: de-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 443–457 (2012)
31. Konoth, R.K., et al.: Minesweeper: an in-depth look into drive-by cryptocurrency mining and its defense. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1714–1730 (2018)
32. Li, S., Kang, M., Hou, J., Cao, Y.: Mining node.js vulnerabilities via object dependence graph and query. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 143–160 (2022)
33. Li, W., Ming, J., Luo, X., Cai, H.: {PolyCruise}: A {Cross-Language} dynamic information flow analysis. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 2513–2530 (2022)
34. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The ghost in the browser: analysis of web-based malware. In: First Workshop on Hot Topics in Understanding Botnets (HotBots 07) (2007)

35. Quarta, D., Salvioni, F., Continella, A., Zanero, S.: Toward systematically exploring antivirus engines. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, pp. 393–403 (2018)
36. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: ACSAC '10: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 31–39 (2010)
37. Romano, A., Lehmann, D., Pradel, M., Wang, W.: Wobfuscator: obfuscating Javascript malware via opportunistic translation to webassembly. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 1574–1589 (2022)
38. Romano, A., Zheng, Y., Wang, W.: Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1129–1140 (2020)
39. Shen, S., Zhu, X., Dong, Y., Guo, Q., Zhen, Y., Li, G.: Incorporating domain knowledge through task augmentation for front-end Javascript code generation. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1533–1543 (2022)
40. Srndic, N., Laskov, P.: Practical evasion of a learning-based classifier: a case study. In: 2014 IEEE Symposium on Security and Privacy, pp. 197–211 (2014)
41. Wang, J., Xue, Y., Liu, Y., Tan, T.H.: JSDC: a hybrid approach for Javascript malware detection and classification. In: ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, pp. 109–120 (2015)
42. Xiao, F., et al.: Abusing hidden properties to attack the node.js ecosystem. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 2951–2968 (2021)
43. Xue, Y., Wang, J., Liu, Y., Xiao, H., Sun, J., Chandramohan, M.: Detection and classification of malicious Javascript via attack behavior modelling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 48–59 (2015)
44. Zhu, S., et al.: Measuring and modeling the label dynamics of online anti-malware engines. In: 29st USENIX Security Symposium (USENIX Security 20), pp. 2361–2378 (2020)