

Wemby's Web: Hunting for Memory Corruption in WebAssembly

OUSSAMA DRAISSI, University of Duisburg-Essen, Germany

TOBIAS CLOOSTERS, University of Duisburg-Essen, Germany

DAVID KLEIN, TU Braunschweig, Germany

MICHAEL RODLER, Amazon Web Services, Germany

MARIUS MUSCH, TU Braunschweig, Germany

MARTIN JOHNS, TU Braunschweig, Germany

LUCAS DAVI, University of Duisburg-Essen, Germany

WebAssembly enables fast execution of performance-critical in web applications utilizing native code. However, recent research has demonstrated the potential for memory corruption errors within WebAssembly modules to exploit web applications. In this work, we present the first systematic analysis of memory corruption in WebAssembly, unveiling the prevalence of a novel threat model where memory corruption enables code injection on a victim's browser. Our large-scale analysis across 37 797 domains reveals that an alarming 29 411 (77.81 %) of those fully *trust* data coming from potentially attacker-controlled sources. As a result, an attacker can exploit memory errors to manipulate the WebAssembly memory, where the data is implicitly trusted and frequently passed into security-sensitive functions such as `eval` or directly into the DOM via `innerHTML`. Thus, an attacker can abuse this trust to gain JavaScript code execution, i.e., Cross-Site Scripting (XSS).

To tackle this issue, we present Wemby, the first viable approach to efficiently analyze WebAssembly-powered websites holistically. We demonstrate that Wemby is proficient at detecting remotely exposed memory corruption errors in web applications through fuzzing. For this purpose, we implement binary-only WebAssembly instrumentation that provides fine-grained memory corruption oracles. We applied Wemby to different websites, uncovering several memory corruption bugs, including one on the Zoom platform. In terms of performance, our ablation study demonstrates that Wemby outperforms current WebAssembly fuzzers. Specifically, Wemby achieves an average speed improvement of 232 times and delivers 46 % greater code coverage compared to the state-of-the-art.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: WebAssembly, Memory Corruption, Taint-tracking, Fuzzing

ACM Reference Format:

Oussama Draissi, Tobias Cloosters, David Klein, Michael Rodler, Marius Musch, Martin Johns, and Lucas Davi. 2025. Wemby's Web: Hunting for Memory Corruption in WebAssembly. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA059 (July 2025), 24 pages. <https://doi.org/10.1145/3728937>

1 Introduction

WebAssembly, or short Wasm, enables safe and efficient execution of native code in browser-based applications [33]. Using ahead-of-time compilation and bytecode optimizations, Wasm promises

Authors' Contact Information: Oussama Draissi, oussama.draissi@uni-due.de, University of Duisburg-Essen, Essen, Germany; Tobias Cloosters, tobias.cloosters@uni-due.de, University of Duisburg-Essen, Essen, Germany; David Klein, david.klein@tu-braunschweig.de, TU Braunschweig, Braunschweig, Germany; Michael Rodler, mrodler@amazon.de, Amazon Web Services, Düsseldorf, Germany; Marius Musch, m.musch@tu-braunschweig.de, TU Braunschweig, Braunschweig, Germany; Martin Johns, m.johns@tu-braunschweig.de, TU Braunschweig, Braunschweig, Germany; Lucas Davi, lucas.davi@uni-due.de, University of Duisburg-Essen, Essen, Germany.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

© 2025 Copyright held by the owner/author(s).

ACM 2994-970X/2025/7-ARTISSTA059

<https://doi.org/10.1145/3728937>

to perform at near parity with native code execution, enabling the deployment of performance-critical applications on the web. Prominent tools and websites using Wasm include *Zoom*, *twitch.tv*, *Adobe Acrobat*, *Disney+*, and *Google Earth*. As of 2024, already 4.4 % of all Google Chrome page loads instantiate Wasm modules [17], and this trend is growing.

Wasm enhances browser sandbox security by ensuring a clear interaction interface between JavaScript and Wasm. In combination with Wasm’s built-in Software Fault Isolation (SFI) properties and structured control flow, it further protects Internet users from memory corruption attacks. This empowers programmers to compile memory-unsafe languages like C and C++ into Wasm and host them on websites, mitigating concerns about code-injection [71] and code-reuse [77, 81] attacks. As evidenced by Hilbig et al. [39], who found that over two-thirds of Wasm binaries were compiled from memory-unsafe languages. However, compiling unsafe code to Wasm does not eliminate the inherent memory unsafety of these languages, and vulnerabilities still propagate to Wasm modules.

The efficiency and legacy code deployment of Wasm open a new potential attack vector: memory errors can corrupt *trusted* host data within the *untrusted* Wasm linear memory. This trusted data, often unsanitized, can range from simple integers to complex media data and can originate from various sources. Functions processing this *external input data* are *security-critical* and serve as entry points for exploitation. Through this interface, an attacker can interact with the Wasm module to inject payloads. Consequently, memory errors in Wasm can lead to the corruption of trusted host data, enabling classic web attacks such as cross-site scripting (XSS) [26, 54, 63].

Remarkably, the exploitation of bugs within Wasm is significantly easier compared to exploiting applications running on modern operating systems. Modern systems increasingly utilize defense-in-depth strategies deploying mitigation techniques such as stack canaries [19] and address space layout randomization (ASLR) [91]. In contrast, Wasm lacks these protection techniques, allowing bugs like stack-based buffer overflows to easily overwrite the entire Wasm memory and reliably hijack the runtime environment.

The discovery of memory errors in native applications is a well-studied research area, using diverse techniques such as fuzzing [8, 23, 34, 62, 96], taint analysis [78, 92, 94], and symbolic execution [3, 6, 12, 13, 16, 32]. Existing proposals to uncover vulnerabilities in Wasm code, using static analysis [11, 86], dynamic analysis [5, 56], taint tracking [27], symbolic execution [37, 38], and fuzzing [36, 58], primarily targeting WASI [95] applications, an extension enabling Wasm modules to run outside the browser. Yet, these methods often yield high false negatives when applied to web-integrated Wasm modules due to their stateful nature and complex interaction with JavaScript. Furthermore, this *context-free* analysis of a Wasm module outside its environment (i.e., the website) results in missed bugs and false positives. This context-free analysis does not consider if an attacker can manipulate the linear memory of the vulnerable Wasm module. Likewise, current vulnerability detection tools are unable to determine if the website properly sanitizes data written to or read from untrusted Wasm memory, or if this data is passed into security-critical sinks or JavaScript functions, which could potentially lead to XSS. As a result, state-of-the-art analysis techniques fall short for Wasm code as it is commonly deployed on the web and highlights a significant gap in research.

Contributions. This paper presents the first large-scale study (Section 4) into a novel—and surprisingly not yet studied—threat model and systematically explores the impact of memory corruption in websites using WebAssembly. Our findings on the Wasm web landscape show that 29 411 websites *trust unsanitized data* from network sources, providing a remote attacker an entry point to inject payloads into a victim’s browser through Wasm, making them susceptible to XSS in the presence of memory errors. Furthermore, 2782 of these websites trust Wasm data by executing it with unsafe APIs like `eval()` or loading data into `innerHTML` without verifying the integrity of this data, which may have been compromised through memory corruption. For example, websites from the BKK

Electronics group (e.g., id.oneplus.com) embed JavaScript code (FingerprintJS [24]) within their Wasm modules and use `eval()` to execute it in the client's browser. An attacker with control over the Wasm memory, could corrupt the JavaScript code, allowing them to execute malicious code within the victim's browser.

To address this emergent threat from memory corruption on the web, we present the design and implementation of *Wemby* (Section 6), an open-source framework for vulnerability detection in Wasm-powered websites. *Wemby* overcomes the limitations of context-free analysis by considering the Wasm module's environment, ensuring reliable detection of memory manipulation and data sanitization issues. Furthermore, *Wemby* automatically analyzes the website's interaction with its Wasm modules. This analysis reveals how an attacker can inject payloads into the Wasm linear memory and whether corrupted data from the Wasm linear memory is loaded into security-critical sinks. Taking the results of the analysis into account, *Wemby* detects vulnerabilities with a fuzzer that does not rely on the existence of source code. Here, *Wemby* uses Wasm memory snapshots to capture the execution state and to ensure bug reproducibility on websites.

For a comprehensive analysis of a Wasm-powered website, we have largely extended the taint-aware Foxhound browser [52] with Wasm-specific sources and sinks. Foxhound is a web browser with dynamic data-flow tracking enabled in the JavaScript engine and DOM, based on Mozilla Firefox. It is widely used in academic research for various purposes, including fingerprinting [2, 10], client-side XSS analysis [51, 72], and client-side Cross-Site Request Forgery detection [48], among others. Our modifications to the Foxhound Browser amount to approximately 2300 lines of code.

Wemby offers fine-grained detection of attacker-controlled data flows (e.g., `location.href`) entering the Wasm linear memory and accurately tracks whether this data is subsequently loaded into security-critical sinks, such as `eval()`. We are not aware of any existing approach that can provide such insights to a developer. Moreover, *Wemby* also integrates a custom-engineered fuzzer. This fuzzer tests security-critical functions identified by *Wemby*'s analysis and uses memory snapshots of the linear memory, ensuring bug reproducibility. This is complemented by a binary-only instrumentation to insert novel bug oracles to identify memory corruption errors effectively. Overall, *Wemby* consists of 5890 lines of code. Combining our instrumented Foxhound browser with *Wemby*'s capabilities in detecting memory corruption allows for a thorough investigation of data flows and a clear understanding of the potential for an attacker to escalate memory corruption bugs into XSS.

We perform a large-scale evaluation to test *Wemby*'s ability to detect security-critical functions and fuzz highly complex websites (Section 8). In addition, we showcase *Wemby*'s capability in detecting a severe bug (CVE-2018-14550) in a web application. This specific bug in the `libpng` library serves as a representative case for Wasm-powered websites [11, 36, 54, 58], as performance-demanding applications such as Zoom and `twitch.tv` utilizes similar libraries from memory unsafe languages for media decoding and leverage Wasm to boost performance. Moreover, we demonstrate how *Wemby* not only detects memory corruption in Wasm-powered websites but also show how these bugs can be elevated to XSS. Although this step often requires significant manual effort, *Wemby*'s holistic analysis of the target website aids in this crucial process. We also conduct an efficiency analysis (Section 9) by fuzzing 76 websites, demonstrating that *Wemby*'s fuzzer largely outperforms related Wasm bug-finding solutions, such as WAFL [36] and Fuzzm [58], by three orders of magnitude in terms of execution speed and covers on average 46% more code.

In summary, we make the following contributions:

- We present the first analysis of a novel threat model and demonstrate its prevalence in the web (Section 4).
- We develop *Wemby*, an open-source analysis tool that can holistically analyze and detect memory corruption in Wasm-powered websites (Section 6).

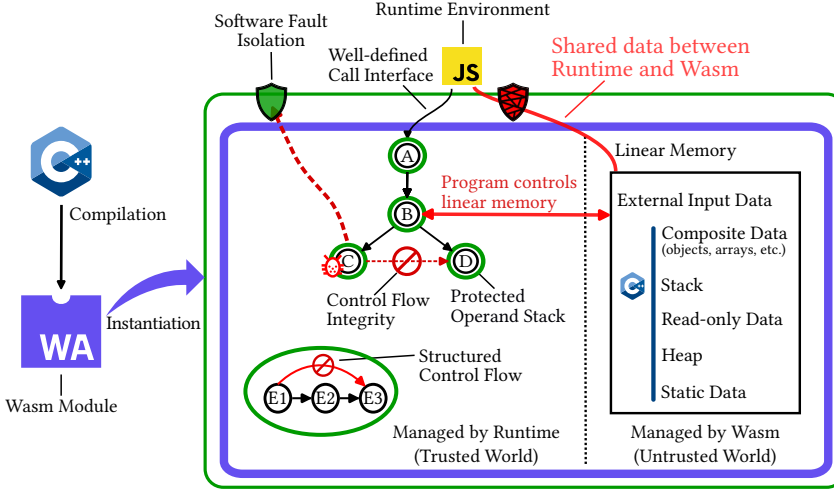


Fig. 1. An overview of the Wasm security model.

- We implement a novel binary-only instrumentation framework to detect memory corruption vulnerabilities in Wasm (Section 7).
- We demonstrate how Wemby detects SFI breaking bugs in Wasm-powered websites (Section 8).
- We show that Wemby outperforms the state-of-the-art Wasm fuzzers [36, 58] by three orders of magnitude while achieving significantly higher precision and code coverage (Section 9).

To foster research on the security of WebAssembly, we release Wemby and our Foxhound fork at <https://github.com/uni-due-syssec/wemby>.

2 WebAssembly Security

WebAssembly [33] is a bytecode standard supplementing the web stack. Since web browsers commonly execute untrusted code loaded from the Internet, Wasm’s design emphasizes its security schemes [65]. Figure 1 details the security model and its main goals: (i) protect users from faulty code, and (ii) provide developers with primitives and defense mechanisms to develop secure applications.

Software Fault Isolation (SFI). Wasm runs in an isolated execution environment, and any interaction between the Wasm module and its runtime is limited to explicitly exported and imported function calls. Hence, the runtime environment can provide functions for the Wasm module and vice versa. Thus, a malicious WebAssembly program cannot arbitrarily access code or data of its runtime.

For additional protection of the runtime environment, Software Fault Isolation (SFI) allows the separation of code and data memory into *trusted* and *untrusted* memory regions. To accomplish this, Wasm operates on multiple isolated memory regions: (i) the code section, (ii) the call stack, (iii) the operand stack, and (iv) the linear memory, which is the only memory region *arbitrarily* accessible by the Wasm program. In contrast to x86 applications, where code and data pages reside in the same address space, a Wasm module’s own code section is neither readable nor writable by its Wasm code. Both the operand stack and call stack are moderated by the runtime; Wasm code can interact with the operand stack through instructions that pop or push operands, and the call stack is implicitly maintained for function calls and returns. Addresses of the runtime environment are never revealed to the Wasm module and cannot be accessed by it. The linear memory is controllable by the Wasm and JavaScript code and addressed using a zero-based index. Thus, Wasm modules are restricted to their private memory, and any

illegal memory access outside the linear memory raises a *runtime error*. This design enables both the runtime and the embedding website to safely compile, instantiate, and interact with untrusted Wasm.

Control Flow Integrity (CFI). Despite the use of potentially memory-unsafe languages in Wasm code, the structured control flow [65] properties protect the runtime from attacks. Moreover, Wasm employs CFI, a well-researched security mechanism [1, 41, 66, 77], which defends against memory corruption attacks by enforcing predefined control-flow graphs for program execution. Wasm's CFI scheme guarantees the integrity of all possible branch targets. Function calls are direct, or indexed from a predefined list, preventing jumps to arbitrary functions or instructions. These indices are known during instantiation and cannot be altered at runtime. Indirect function calls are also subject to type signature checks, ensuring control-flow adherence. Furthermore, control flow instructions, such as unconditional and conditional branches, are bound to code blocks, restricting exits to enclosing blocks. This differs from x86, where branches can target arbitrary addresses. This design assures uniform operand stack changes for every control-flow path, preventing corruption by unexpected operations. Function return addresses remain secure as the call stack is isolated from Wasm instructions. However, these properties protect the runtime, and bugs in custom code can still corrupt application data.

3 Threat Model: Wasm in the Browser

Compromising a user account typically involves a server exploit, or stealing the credentials through phishing, or XSS, which is the focus of our work. We introduce a novel method to achieve XSS by exploiting Wasm modules that suffer from memory corruption bugs. An attacker can exploit the fact that Wasm modules integrated into websites often process *untrusted* data combined with *trusted* data. A notable scenario involves a web server passing user-generated data to other users, as shown

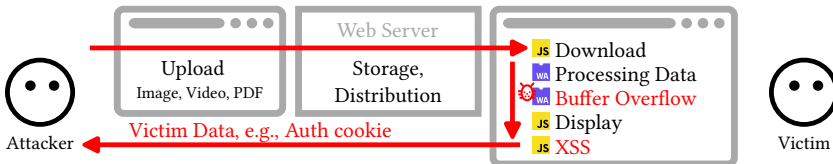


Fig. 2. Exemplary attack flow: An Attacker uploads media data that contains an exploit. When the victim visits the website, the exploit triggers a bug in the Wasm code that leads to XSS and the attacker gaining, e.g., the authorization cookie of the victim.

in Figure 2. One potentially malicious user, the attacker, generates data that is distributed via the website's server and processed by the other client using Wasm. The Wasm code decodes the data and renders it into HTML, which is subsequently inserted into the DOM for display. An application model that maps to these scenarios is found in videotelephony (e.g., Zoom) and shared media platforms (e.g., twitch.tv) that display the data generated by other users.

In the presence of memory errors in the Wasm module, the attacker can hijack the decoding process and provide arbitrary HTML code for insertion, effectively providing the attacker arbitrary JavaScript code execution. Consequently, even without a direct XSS vulnerability, the attacker can still gain JavaScript execution by exploiting errors in Wasm modules. This underscores the evolving landscape of web security threats and the need for robust defenses against Wasm-specific vulnerabilities.

3.1 Motivating Example

We further emphasize the importance of discovering memory corruption in Wasm with a motivating example inspired by related work [11, 36, 54, 58] and existing web applications (e.g., Figma, Zoom,

```

1 class Converter {
2   public: string pnm2png(unsigned char []) { };
3 }
4
5 Converter *converter = nullptr;
6 // 1. Initializes the converter
7 void initialize_converter() {
8   converter = new Converter(); }
9
10 // Other functions that hinder analysis
11 string some_func_01(byte* src)
12 // ...
13 int some_func_0X(int x, float y, byte* src)
14
15 // 2. Convert and show image inside the browser
16 string convert_and_preview(byte* src) {
17   if (converter) {
18     // 3. Stack-based buffer overflow in pnm2png
19     string img = converter->pnm2png(src);
20     // 4. An attacker can overwrite this string
21     write_to_dom(img); // document.write(img)
22     return img;
23   }
24 }

```

Fig. 3. C++ source code of an Wasm module using a vulnerable libpng library. An attacker can escalate this vulnerability to execute arbitrary JavaScript.

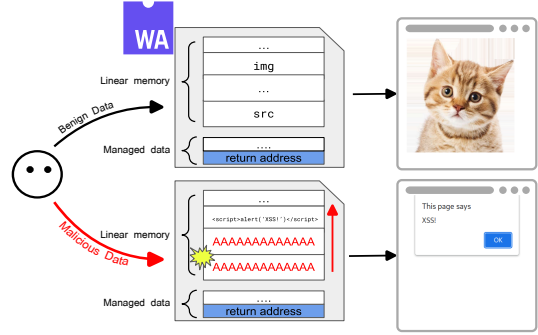


Fig. 4. View of the linear memory of the motivating example. A buffer overflow can corrupt all static and dynamic data in the linear memory.

and twitch.tv). Concerning Figure 2, consider an image-sharing service where users can upload an image into a registry. Other websites can embed these images by using the provided JavaScript framework, which loads the images from the registry service. On the client side, this framework includes Wasm to efficiently convert images from, e.g., PNM to PNG before displaying them. As we will demonstrate in our evaluation (Section 4), this is a very typical use-case for deploying Wasm. For example, Google CanvasKit [29] and Amazon IVS [4] both offer packages, containing JavaScript and Wasm, to facilitate embedding third-party content on websites.

Figure 3 describes an example JavaScript and Wasm interface for this kind of application. On the client side, after the `converter` object is initialized (line 8), the `convert_and_preview` function uses `libpng` to convert a received PNM to a PNG image and renders it directly to the DOM. This application is vulnerable due to a buffer overflow vulnerability in this version of `libpng` (CVE-2018-14550).

Figure 4 describes the layout of the linear memory during the image conversion. In benign cases, `src` (line 16) does not overflow, and the program appends the converted image `img` (line 19) to the DOM. However, the lack of memory protection inside the linear memory allows an attacker to perform a stack-to-heap buffer overflow and overwrite the HTML template to display the image. In this scenario, the attacker uploads an XSS payload to the image registry. An unsuspecting user might request this image, inadvertently loading the attacker's payload into their memory, resulting in an XSS attack.

Note that traditional attacking techniques that overwrite the return address are ineffective on Wasm, as the return address is located outside the linear memory, making it inaccessible to the overflow. Conversely, Wasm does not support standard defense mechanisms like guard pages, as its linear memory model lacks segmentation and memory permissions. This limitation makes it difficult to isolate and safeguard different memory regions effectively.

3.2 WebAssembly (In-)Security

The fundamental question is whether the security mechanisms deployed in Wasm are sufficient to prevent exploitation. Control flow and data flow of Wasm modules are restricted. Access to the host environment is limited to explicitly exported functions. An attacker cannot directly access return addresses to hijack the control flow because these are inaccessible from inside the Wasm module. However, code vulnerabilities do not vanish but still exist given that Wasm code is developed mainly in memory-unsafe languages [39], such as C and C++.

Coarse-Grained CFI Checks. We first study the CFI security model. Wasm deploys signature type checking for indirect function calls. However, these checks are confined to Wasm's limited set of types, using only two integer types (i32 and i64) and two floating-point types (f32 and f64). This restriction stands in contrast to more comprehensive CFI checks found in languages like C and C++, which can validate function signatures containing user-defined or compound types. As a result, the signature checks performed by Wasm are considered highly *coarse-grained*, i.e., multiple functions are mapped to the same signature type. As existing work in this field has shown [21, 26, 28, 63], an attacker can easily divert the control-flow to a large pool of possible functions without violating the function signature.

Insecure Memory Management. A typical executable (e.g., Linux ELF or Windows PE) divides memory into different regions, such as the stack and the heap. The pages in these memory regions are assigned explicit memory permissions, e.g., data regions are writable or read-only but not executable. However, when compiled to Wasm, all these different non-executable memory regions reside in a single continuous array of bytes, the so-called *linear memory*. Linear memory is represented as a JavaScript array, making it mutable and effectively both readable and writable. In addition to the stack and heap, the linear memory contains the read-only data sections of the program and its linked binaries, such as the `libc`. As a result, the isolation and memory permissions typically utilized for ordinary executables no longer exist inside the linear memory. For instance, read-only memory regions become writable when compiled to Wasm. While there is no official documentation explaining this decision, we speculate this decision has been made for performance reasons, i.e., enabling efficient memory management and direct manipulation which is vital for high-performance applications. This design pattern also promotes consistency with JavaScript, facilitating interoperability and effective use of typed arrays and buffers.

Reliable Exploitation. While the Wasm security model mitigates code injection [71] and code reuse attacks [81], data-only attacks [41, 42] are feasible in Wasm [26, 54, 63]. These attacks circumvent CFI because they do not rely on deviating the intended control-flow of a program. Typically, data-only attacks involve multiple stages, such as exploiting a memory leakage vulnerability to perform a memory disclosure attack [74, 83], which is necessary to bypass Address Space Layout Randomization (ASLR) [91]. These additional stages increase the complexity of the attack. However, the static placement of data in Wasm and the absence of randomization within the linear memory actually simplify the exploitation of memory corruption vulnerabilities. In fact, the memory addresses of the data that an attacker seeks to manipulate reside at predictable addresses, which allows for reliable and reproducible exploitation.

4 Prevalence of a New Threat in Wasm-Powered Websites

Given the threat model and vulnerable application patterns described in Section 3, we derive four properties (P) of a website required to be exploitable via Wasm memory corruption: (P1) the attacker can manipulate the content of the linear memory of the vulnerable module, (P2) the website performs no sanitization of the data written into and from the linear memory, and (P3) the website reads from the linear memory and passes this data into a JavaScript function that can lead to code execution. Lastly, (P4) the website deploys a Wasm module with a memory corruption vulnerability.

To show that the presented threat model occurs in the wild, we measured the three properties related to websites using Wasm modules, i.e., P1 to P3. Based on these insights, we will assess P4 in the following sections, as this primitive presents its own set of challenges. While TaintAssembly [27] and, to some extent, Wasabi [56] enable taint tracking within Wasm, neither solution currently supports the tracking of data flows across boundaries involving Wasm, JavaScript, and the DOM. Moreover, the identification of attacker-controllable input sources (e.g., `location.href`) is performed

Table 1. Top sources loaded into the linear memory.

Source	# Dataflows	# Domains
HTMLCanvasElement.toDataURL	154 147 826	25 117
WebSocket	123 738 655	4971
XMLHttpRequest.response	16 333 705	5118
sessionStorage.getItem	269 892	41
localStorage.getItem	251 169	160
location.href	124 652	2401
document.baseURI	69 674	2065
document.cookie	14 474	200
document.referrer	9143	334
location.pathname	6285	194
document.documentURI	5175	26
location.search	3050	41
location.hash	1728	20

Table 2. Top sinks receiving data from linear memory.

Sink	# Dataflows	# Domains	Vulnerability
Function.ctor	386 745	2286	XSS
innerHTML	38 742	842	XSS
a.href	23 993	1341	XSS
eval	9297	643	XSS
fetch.url	5045	1054	Request Hijacking
sessionStorage	2524	345	Stored XSS
localStorage	1437	317	Stored XSS
XMLHttpRequest.setRequestHeader	1417	68	Request Hijacking
XMLHttpRequest.open(url)	1384	497	Request Hijacking
script.src	917	266	XSS
window.postMessage	578	97	XSS
XMLHttpRequest.send	328	123	Request Hijacking
iframe.src	174	113	XSS
document.cookie	123	80	Stored XSS
WebSocket	224	127	Request Hijacking

manually, underscoring the need for a new approach. To this end, we modified the open-source taint browser project Foxhound [52], a Firefox fork that supports tracking of tainted values through both the JavaScript engine and the DOM. We added support to track data flowing *into* and *out* of the Wasm linear memory. We analyzed how JavaScript interacts with the Wasm linear memory and found that it typically uses the `TypedArray` and `DataView` data types. Based on this analysis, we enhanced the Foxhound browser by appending taint information to the base class for all array buffer views, `ArrayBufferViewObject`, and all relevant subclasses. Additionally, we modified methods, such as `Uint8Array.set()`, to propagate tainted values when using these objects. In summary, we added 2343 lines of code across 118 files to the Foxhound project. With these changes, we can track data flows from attacker-controlled sources (e.g., URLs, WebSockets) into the Wasm linear memory. Similarly, by tainting data originating from linear memory, we can measure whether there are data flows to security-critical sinks, e.g., `eval` or `innerHTML`.

Measurement Parameters. We visit the websites assembled from [17] and randomly choose three links from the front pages to follow. After loading each page, Foxhound automatically analyzes the interactions between JavaScript and Wasm for 30 seconds. In line with prior research [7, 48, 50, 64, 87], our crawler does not perform any user interactions. In total, we inspected 205 977 websites, of which 37 797 use Wasm on load.

Data Flows Into and From Linear Memory (P1). Our analysis of the Wasm landscape revealed that 29 411 websites load data from attacker-controlled sources into the linear memory, effectively allowing an attacker to inject her payload into a victim’s browser. Table 1 shows the different origins of the data. Note that these contain sources not considered for classical web attacks, e.g., XSS. On websites using Wasm, data received via WebSockets or displayed in canvas elements often originates from other users or servers. For example, during a Zoom call, the video feed from other participants is received over a WebSocket and then loaded into each participant’s Wasm linear memory.

Wasm Modules Rely on Dangerous APIs To Interact With the Environment (P2). Although the security model of Wasm could be used to build a fine-grained host API that protects the DOM from malicious modifications, our study indicates that developers rely on generic APIs that provide Wasm modules with extensive DOM access. While access to `eval` may be required for third-party and legacy modules that cannot be changed or performance-critical modules that are slowed down by a tight API, this makes `eval` readily available to an attacker to escalate memory corruption in Wasm modules to XSS. Table 2 summarizes our findings: Overall, we learned that 4065 websites expose their DOM to the Wasm module. Most Wasm compilers provide developers access to JavaScript API’s. For instance, if the module creates JavaScript functions or objects, Emscripten automatically exposes functions

such as `Function.ctor` or `eval` to the Wasm module. Based on the documentation [22], this is not fully clear and exposes the module to the strongest XSS primitives available. These functions provide access to *anything* in the JavaScript environment—including cookies, `sessionStorage`, and the DOM.

One particular website, queenscommonwealthtrust.org, first iterates through an array in the Wasm linear memory containing strings like `document`, `window`, and `location`. It then calls into JavaScript, which uses `eval` on each item in these objects and stores the results back into the Wasm memory. However, this *trust* in the integrity of the data stored in the Wasm linear memory introduces a new security threat, as the data used in these APIs comes from linear memory and can be manipulated in the presence of memory errors.

Concealing JavaScript in Wasm. During the analysis of the websites that use these unsafe JavaScript APIs, we discovered another motivation for developers to include `eval` in the API provided to Wasm: The websites of the BKK Electronics Group use Wasm to store an obfuscated version of *FingerprintJS* [24] in their linear memory, which is later deobfuscated and loaded into the browser. This practice highlights the trend of concealing JavaScript code within Wasm modules, allowing them to disguise the use of a well-known fingerprinting script and circumvent traditional blocking mechanisms. The process works as follows: id.oneplus.com instantiates a Wasm module upon page load, containing encrypted JavaScript code in its memory. After instantiation, the JavaScript code is decrypted, enabling the website to load this data and use it as input for `eval()`. Nevertheless, this obfuscation does not protect users from memory corruption. The static placement of decrypted data in linear memory makes it easier to exploit memory errors, which can overwrite the result that is fed into `eval`, resulting in XSS. This further emphasizes the significant impact of Wasm bugs and the role of Wemby in detecting them.

Sanitization (P3). Web applications increasingly rely on Wasm modules for DOM manipulation, local and session storage, and web requests, with analytics platforms often sending tracking data through these requests. However, attackers can exploit this by corrupting URLs used to fetch resources, injecting malicious JavaScript into a victim's browser. The level of risk varies depending on the website's *trust* in and use of data within Wasm's linear memory. Our analysis reveals that developers rarely preprocess tainted data loaded into Wasm memory, particularly on client-side JavaScript. This lack of data sanitization is widespread: 98.48% of data flows into Wasm linear memory are unsanitized, enabling the exploitation of memory errors. Moreover, 97.30% of outgoing data from Wasm into security-critical JavaScript APIs or HTML sinks is also unsanitized. Specifically, 95.28% of the websites we studied demonstrated data flows where tainted data entered Wasm memory without modification and was subsequently used in security-sensitive contexts, emphasizing the significant security risks associated with unprocessed Wasm data flows.

Summary. Overall, 95.28% websites in our large-scale study provide an attacker with the possibility to inject a payload into a victim's linear memory while trusting the data read from linear memory sufficiently to pass it into security-sensitive sinks without further sanitization. An attacker exploiting a memory corruption error in a Wasm module can leverage this trust to escalate the memory corruption into an XSS vulnerability. In conclusion, we have shown that the threat model described in Section 3 directly applies to the vast majority of Wasm-powered websites.

5 Web(Assembly) Challenges For Fuzzing

Reviewing the final property P4, which involves detecting memory errors, necessitates a novel approach for Wasm-powered websites. Existing approaches to detect memory errors in Wasm modules [11, 36–38, 58] focus on WASI [95] applications. WASI is an extension for Wasm enabling to run Wasm binaries outside the browser while providing a common set of system calls to interact with the underlying system, e.g., for file access. However, Wasm modules in web applications rely on

their integration with the website's JavaScript code to communicate with their environment. While a *context-free* analysis of Wasm modules might uncover some bugs, taking the JavaScript integration into account is essential to accurately assess the attack surface and vulnerabilities of Wasm modules in web applications. In the following, we detail the issues existing context-free approaches face on the example from Section 3.1.

Challenge 1: Wasm and JS Interaction Interface

Unlike WASI applications with a single entry-point, Web-Wasm modules allow JavaScript to call *any* exported Wasm function at *any* time. For instance, in the motivating example, by only examining the Wasm module, it is impossible to determine which functions are *actually* used. A context-free fuzzer might find a bug in `some_func_01()` (line 11 in Figure 3). However, if the website's JavaScript code does not use this function, this bug is a false positive. Without a thorough understanding of the application, an analyst might test every function, consuming substantial time. This approach does not scale, as the number of functions in a Wasm module can be large; for instance, Zoom uses 18 765 different Wasm functions during a video call.

Besides knowing which functions are called, it is essential to understand the specific data and semantics that each function requires. For example, a vulnerable function expecting a pointer to a buffer might crash if given incorrect data, leading to false positives without detecting the underlying vulnerability. For instance, the vulnerable function in the motivating example expects a pointer to a buffer. If an incorrect data type is provided to the function, it could crash due to these incorrect semantics, e.g., by accessing data outside the linear memory, but the underlying vulnerability might not be detected. Ultimately, this results in a false positive.

Challenge 2: Statefulness

Wasm modules often require a specific memory state to execute function calls correctly. This means that existing approaches must initialize and manage the memory state accurately for meaningful testing. If the memory is in an unexpected state, function calls may fail or behave unpredictably, leading to false negatives, i.e., missed vulnerabilities. In particular, Wasm memory is often initialized using `memcpy` or `Array.set` functions for data processing or object construction before the actual function calls. For instance, the vulnerable function in Figure 3 is only called if the `converter` object is initialized (line 8). Consequently, detecting the bug in `libpng` depends on the state of the linear memory.

Similarly, Wasm modules can maintain internal state across multiple function calls. This state can then affect the behavior of subsequent function calls, making it essential for bug-finding approaches to preserve and manage this state during testing. For instance, a function might behave differently based on the results of previous function calls. Context-free attempts to find bugs without considering the internal and memory states can lead to incorrect conclusions about the module's behavior and potential vulnerabilities, resulting in incomplete or inaccurate testing.

Challenge 3: Detecting Memory Corruption in Wasm and Test-Driver Code Generation

Detecting memory corruption in Wasm modules during runtime is limited to identifying Wasm *traps* [65], which are triggered by invalid operations such as out-of-bounds memory access of the linear memory. While these traps ensure that memory operations stay within allocated boundaries, they cannot detect overflows *within* the linear memory, thereby missing vulnerabilities like buffer overflows. This is critical for compiled languages emulating a stack with linear memory (e.g., C and C+). While WAFL [36], SeeWasm [37], and Wasmati [11] rely on traps for error detection, Fuzzm [58] instruments Wasm modules to detect heap and stack buffer overflows, addressing spatial memory safety violations. However, this approach is limited to WASI-Wasm modules and lacks temporal

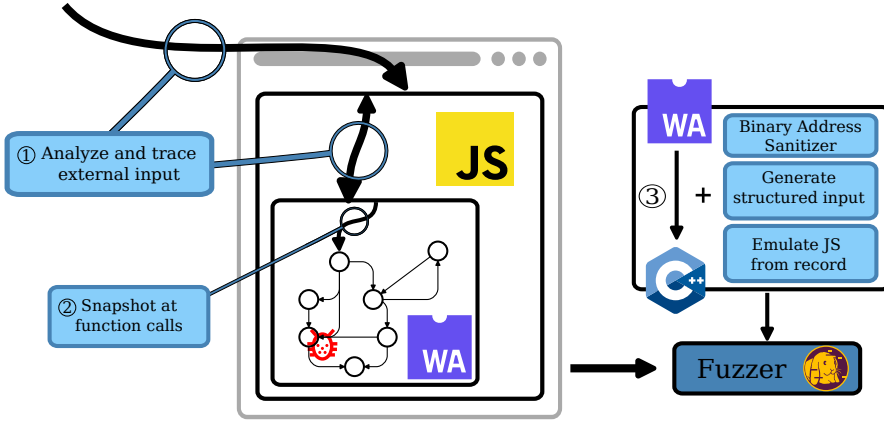


Fig. 5. Architecture design of Wemby. Our tool analyzes the data flow through a web application and records all necessary data to simulate the execution environment for fuzzing using custom bug oracles.

memory safety detection (e.g., use-after-free). Enhancing bug oracles for finer-grained detection is essential for accurately identifying such vulnerabilities.

Unlike WASI-Wasm, which uses standard input (stdin), Wasm modules on the web require structured input. Structured input involves complex data types and relationships between different pieces of data. WASI-Wasm fuzzers can simply use a single input for the entry-point, but for Web-Wasm modules, each corresponding function needs to be associated with its respective structured input. Developing methods to automatically infer or specify the required input structures is crucial for effective bug detection in Wasm modules, ensuring accurate and meaningful testing.

6 Design

Wemby's goal is to automatically find memory corruption bugs in Wasm modules embedded on websites. Our design choices ensure that Wemby tackles each challenge and guarantees that each bug finding is reproducible. The high-level design of our Wasm analysis architecture is visualized in Figure 5: The main idea of Wemby is to: ① collect Wasm function *call traces*, analyze this information and identify security-critical functions (Challenge 1), ② record *snapshots* of websites (Challenge 2), and ③ finally, instrument the Wasm with memory error oracles and find bugs through *fuzzing* (Challenge 3). In the following, we describe how each aspect solves a different set of challenges described in Section 5.

Analyze External Input ①. We analyze the attack surface by tracing every interaction between the website and the Wasm module. Following the threat model in Section 3 and addressing Challenge 1, we investigate how attackers can influence data passed to the Wasm module. Exploitation entry points, such as URLs, are limited to specific communication channels (Table 1), that Wemby classifies these as *external input data*. Note that data coming from network sources are classified as external input data if they originate from a different domain than the host website—e.g., scheck.se processing media data from twitch.tv. The analysis of recorded taints is highly extensible, allowing analysts to implement various de-sanitization schemes. However, as shown in Section 4, many websites either do not sanitize external input, making it a significant attack vector. Wemby captures runtime data from live websites using Wasm by tracing every interaction between the website and the Wasm module. This reveals the data flow between JavaScript and Wasm, detailing which Wasm functions are invoked, their arguments, and the overall module state. This analysis also helps infer which data from Wasm is loaded into security-critical sinks (Table 2), allowing to escalate issues found with Wemby into impactful bugs. Lastly, Wemby uses this interaction data to infer function argument data

types (Challenge 1 and Challenge 3), ensuring accurate structured input for fuzzing. Analyzing these call traces and function arguments enhances fuzzing accuracy and prevents false positives, e.g., due to calling functions with incorrect data types. We evaluate the impact of this analysis in Section 9.1.

Record Snapshots ②. Wasm itself is also stateful, as the linear memory changes throughout the lifetime of the Wasm instance. The linear memory can either be modified by the Wasm module itself or by the website's JavaScript code. A common pattern is for the website's JavaScript to first load data, for example, strings or objects, into the linear memory for the Wasm module to use. To understand these state changes, we take *snapshots* of the linear memory at the beginning of each Wasm function invocation and before it terminates (2). Afterward, Wemby analyzes these snapshots to automatically generate realistic test-driver code. That is code that can reproduce a website's Wasm function call without losing its execution context. This ensures that function invariants, such as data being stored at specific addresses or object initialization, are not violated. This analysis significantly improves the code coverage achieved by Wemby, which we show in Section 9.2.

Fuzzing Web-WebAssembly Natively ③. After analyzing and evaluating the call traces, we can start to fuzz the target. The analysis enables Wemby to fuzz Wasm outside its host environment, i.e., the browser. Thus, we can omit the original Wasm embedding and instead re-compile the Wasm module into an instrumented x86 binary. This allows us to take advantage of state-of-the-art binary optimization and advanced fuzzing techniques [25] to increase the performance of our fuzzer and achieve better results. Previous work [36, 58] uses VM instrumentation, which is significantly slower, as we will show in Section 9.3. Given Wasm's lack of inherent memory protection for linear memory (as discussed in Section 3), Wemby implements Wasm-specific test oracles. These oracles, based on the ASan memory error detection algorithm [80], are adapted to the challenges posed by Wasm modules. Since we target arbitrary websites often employing proprietary Wasm modules, we lack access to their source code. Consequently, our implementation directly targets Wasm binaries and does not require access to the module's source code.

7 Wemby

In this section, we detail the implementation of Wemby, which includes a live analysis of websites. We instrument Wasm modules for call traces, browser interfaces for network traces, and create Wasm linear memory snapshots. Furthermore, we implement fine-grained memory error oracles for Wasm to detect temporal and spatial memory corruption. Unlike clang ASan, Wemby operates on bytecode. Finally, we incorporate the analysis and the instrumented Wasm into the fuzzing process. The engineering effort behind Wemby is substantial, with the automated fuzzer test-driver code generation and memory corruption oracles implemented in C and Python comprising 3400 lines of code, the tracing of Wasm calls in JavaScript consisting of 1900 lines of code, and trace analysis in Rust with 590 lines of code.

7.1 Wasm Website Tracing

While TaintAssembly [27] provides taint tracking within Wasm modules, it does not address the complex interactions between JavaScript, Wasm, and the DOM required by our threat model. Our taint-aware extensions to the Foxhound browser [52] bridge these execution contexts, yet neither approach identifies functions that *actually* access attacker-controlled data. To isolate *security-critical* functions and to evaluate the state of the website, we leveraged the browser automation framework Playwright. Wemby automatically collects the user interactions and the corresponding effects on the Wasm module. With the combination of this taint-aware browser and our Wasm instrumentation, Wemby provides unique insights on Wasm modules and how an attacker can manipulate them.

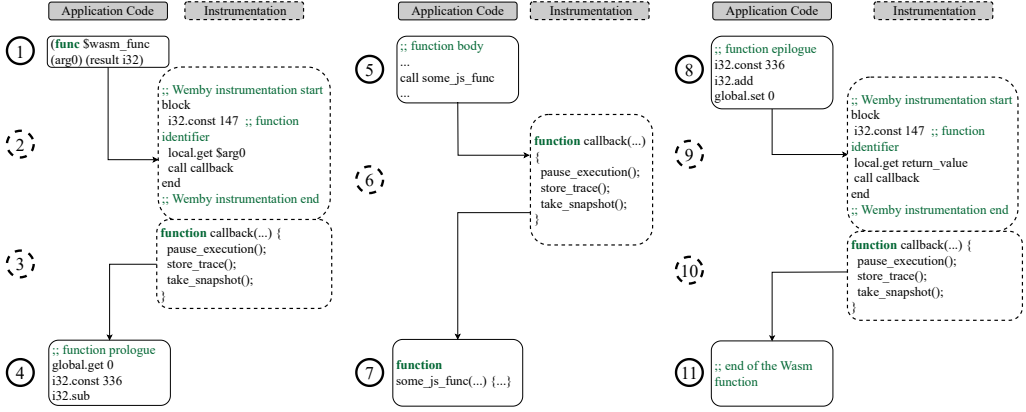


Fig. 6. Dynamically instrumenting Wasm modules in the browser. Solid circles depict the application code and dashed one's instrumentation code.

Wemby *intercepts* loading of the requested Wasm module before it is used by the website. Afterward, Wemby takes the Wasm module and the imported JavaScript functions, i.e., those that Wasm functions can call, and instruments each Wasm function and JavaScript function such that a `callback` function is called at the beginning and at the end of each invocation. We developed our own tailor-made, Wasm instrumentation tool that inserts the callbacks to JavaScript. Our Wasm instrumentation framework is based on *Walrus* [93] and is compiled to Wasm. Figure 6 describes the instrumentation: Wemby inserts a new block at the beginning and end of *every* Wasm function. The block at the beginning of the function (2) pushes the function's identifier and parameters onto the stack. Then, the callback function (3) is called, and the execution of the Wasm function continues (4). This enables Wemby to collect detailed runtime information of the Wasm module, e.g., to record which functions are called, their parameters, and the order of the calls. This instrumentation ensures Wemby logs every call in the Wasm module and stores these traces, thereby capturing the exact interaction of the website with the module's API. Afterward, Wemby snapshots the Wasm linear memory to correctly replicate the application's state during fuzzing (Section 7.3). Instrumenting both Wasm and JavaScript functions (5)-(7) gives us complete insight into the interaction between the two environments. Likewise, we instrument the epilogue in steps (8) to (11) to capture the result of the Wasm function and the state of the linear memory. Finally, Wemby uses the patched Wasm module and JavaScript functions to instantiate the Wasm module.

7.2 Analyzing Traces

Using the call traces to Wasm functions that the tracing (Section 7.1) collected, Wemby analyzes the potential attack surface and determines security-critical functions. These are Wasm functions that are called with attacker-controlled data provided by Foxhound (Table 1). To accomplish this, we analyze each called function's parameters to determine their respective *trust level*. Figure 7 illustrates how Wemby measures the trust level: We approximate a parameter's trust level by performing taint inference [79] against external input data (e.g., `WebSocket` payloads or `location.href`) collected during the tracing. If the trust level of the data passed as a parameter is high, i.e., no data from external sources is directly used, an attacker has no control over this parameter. Consequently, Wemby does not *mutate* parameters with high trust level during fuzzing. If all function parameters have high trust levels, Wemby marks this function as *uninteresting*, as an attacker could not exploit this function. Wemby also infers dependencies between function parameters. For example, it identifies when a

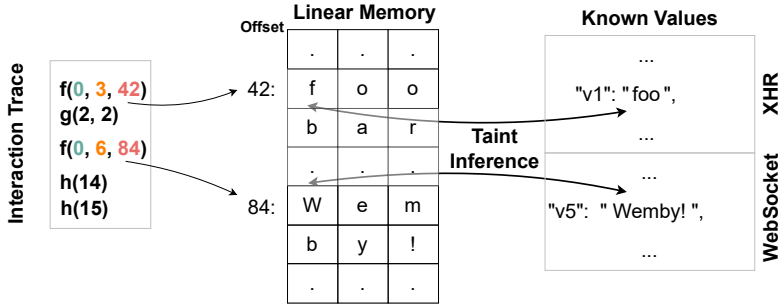


Fig. 7. Example on how Wemby decides what values should be mutated during fuzzing. Parameters are color-coded to reflect the outcome of the analysis. Red is the lowest trust, as the taint inference found a match against external data sources, depicted under Known Values (Table 1). Orange highlights dependent values; here, the length of the strings passed as the next argument. Green are constant values over several invocations.

string is in one parameter and its corresponding size is in another by analyzing the function calls throughout the analysis.

7.3 Fuzzing Web-WebAssembly

Wemby's fuzzing component is designed to realistically resemble the actual usage of the Wasm code in the browser. Here, Wemby automatically generates test-driver code for each security-critical function with its required structured input. Using the call trace analysis results, we can ensure that a bug reported by the fuzzer can also be triggered (Section 9.1) on the website. Further, Wemby re-compiles the Wasm module to native x86 code using *wasm2c* [31] and adds additional instrumentation in this step. This approach allows us to use state-of-the-art native fuzzing techniques to fuzz Wasm modules and omit the (browser) environment, which heavily reduces the overhead and increases the fuzzing performance (Section 9.3).

Memory Error Detection. We implement specific oracles inspired by AddressSanitizer (ASan) [80] to detect memory errors (Section 3) during fuzzing. ASan is a widely used memory error detector for C and C++ programs that maintains shadow memory to track the state of each memory byte, marking them as either accessible or poisoned (redzones). It instruments memory accesses at compile-time to check if they touch poisoned regions, thereby detecting various memory corruption bugs.

While existing WebAssembly fuzzers like Fuzzm [58] rely solely on basic stack and heap canaries for memory error detection, our approach implements a more comprehensive, binary-only version of ASan's capabilities. Operating directly on compiled Wasm modules without requiring source code access, our oracles maintain shadow memory to monitor both the program's data structures and the redzones between them. This enables detection of both spatial memory violations (e.g., buffer overflows) and temporal issues (e.g., use-after-free).

We achieve this comprehensive memory monitoring by instrumenting function prologues and epilogues to track stack allocation sizes, while also hooking the Wasm module's dynamic memory management functions (e.g., `malloc` and `free`) to maintain heap allocation states. By checking if any Wasm instruction attempts to access a redzone, Wemby can effectively detect memory errors. This approach enables both precise detection of memory-related vulnerabilities and the generation of detailed bug reports for invalid memory accesses.

Replicating Wasm Function States and Emulating JavaScript Functions. To automatically replicate the state of the original Wasm function call, Wemby loads the recorded snapshot before calling the target function. WAFL [36] employs a similar technique for fuzzing WASI-Wasm modules by utilizing an initial snapshot. However, their method relies on a single snapshot taken prior to

invoking the entry point of the WASI-Wasm module, used solely to avoid reinitializing linear memory for each fuzzing iteration—an approach that is not feasible for Web-Wasm because of its stateful nature (see section 5). In contrast, our approach leverages snapshotting to accurately replicate the complete state of Web-Wasm for *each* security-critical function. This detailed replication not only ensures reproducibility of bugs in a browser environment but also facilitates a more fine-grained, function-level analysis of problematic behavior. Further, the fuzzer only mutates data of low trust level (Figure 7), while data with a high trust level, e.g., constant data, use the recorded values from the website traces. The fuzzer mutates integer types directly, and for pointers, it allocates buffers with fuzzing input in the Wasm linear memory. Wemby uses the recorded JavaScript function call traces to emulate the environment. We build a map of the arguments use in functions calls to the return value. When the fuzzer calls an imported function, we look up the recorded return value.

Performance Optimizations. Wemby's fuzzer uses AFL++ [25] to take advantage of state-of-the-art fuzzing techniques, such as *persistent mode fuzzing*, which avoids the bottleneck of forking a new process for each fuzzing iteration. Further, we use `mmap` to load the snapshot of the linear memory to optimize the state reset utilizing the operating system's copy-on-write mechanism. This allows us to efficiently reload even 64 MB-sized snapshots with minimal overhead (Section 9.3).

8 Detecting Wasm Sandbox-Breaking Bugs with Wemby

In this section, we use Wemby to evaluate the results of our design choices. As we will see, Wemby is the only existing tool that can find SFI breaking bugs in websites. Here, Wemby is effective in identifying security-critical functions and detecting attacker-controlled input data. It can uncover multiple bugs discovered through our analysis and assist in writing exploits. Ultimately, Wemby is the only approach capable of revealing the presence of P1 to P3 (Section 4) in a website, demonstrating its benefit over related work [11, 36–38, 58].

Ethical Considerations and Vulnerability Disclosure

Since our threat model includes third-party systems, we must consider the potentially harmful effects of our analysis on their infrastructure. Exploiting bugs would involve introducing harmful data into live systems we do not own, which is against ethical guidelines and ACM submission policies. For example, manipulating video streams to trigger memory corruption errors via [twitch.tv](https://www.twitch.tv) could trigger unknown bugs in their server infrastructure. Hence, we skip the servers and substitute the trusted external input data with our payload only on our local machines to escalate bugs to XSS. Similar to previous work [15, 88, 89, 97], we try to contact website authors to report vulnerabilities detected by Wemby. Despite earnest attempts, the lack of readily available contact information hindered progress, forcing us to contact generic email addresses such as `webmaster@domain` [20]. We are currently disclosing bugs and offering collaboration to affected website operators to fix any uncovered vulnerabilities.

8.1 Detecting CVE-2018-14550

We evaluated the accuracy of Wemby by analyzing the web application from Section 3.1. The Wasm module compiled using Emscripten incorporates 442 functions. Wemby reported that 8 of the 442 functions were called during the experiment. These 8 functions were called 82 times, most of which are used for memory management (e.g., `malloc` and `free`) and to copy data into the linear memory. Afterward, Wemby further analyzed these 82 calls. It correctly identified `convert_and_preview` as the sole function processing external input data, namely the PNM image. Furthermore, we learned that two security critical sinks use data from the linear memory: `Function.ctor` and `document.write`.

Wemby generated the fuzzing harness to simulate the execution of the web application at the time of the function call to `convert_and_preview`. This includes the content of the linear memory

```

1  BUG FOUND THROUGH WEMBY ORACLES
2  =====
3  BACKTRACE
4  libpng_fuzz[0x213519]report_error
5  libpng_fuzz[0x2136fb]is_poisoned
6  libpng_fuzz[0x21a3e6]i32_store8
7  libpng_fuzz[0x22bfb]w2c_get_token
8  libpng_fuzz[0x228722]w2c_pnm2png
9  libpng_fuzz[0x216c82]w2c_convert_and_preview
10
11 REASON: STACK COOKIE OVERWRITE!

```

```

1  ==109708==ERROR: AddressSanitizer:
2  stack-buffer-overflow on address 0x7ffc0e9467a0
3  at pc 0x5613223c6229
4  bp 0x7ffc0e9461a0
5  sp 0x7ffc0e946198
6  WRITE of size 1 at 0x7ffc0e9467a0 thread T0
7  #0 0x5613223c6228 in get_token
8  #1 0x5613223c28c6 in pnm2png
9  #2 0x5613223c203e in main
10 #5 0x5613222dd274 in _start
11 ==159476==ABORTING

```

Fig. 8. Excerpt of Wemby’s report for CVE-2018-14550. Fig. 9. Excerpt of clang ASan report for CVE-2018-14550.

before the target function’s execution, which we collected during the analysis. In this example, this places the converter object, used to convert the image by the application, in the linear memory at the correct location. Thus, using the snapshot during the fuzzing ensures the condition in line 17 of Figure 3 is satisfied, as it is when interacting with the website through the browser. In contrast to Wemby, existing Wasm fuzzers [36, 58] cannot satisfy this condition, as they lack insight into the module’s environment and initialization logic. Therefore, these tools are unsuitable for detecting bugs on Wasm-powered websites.

We fuzzed this program for 24 h and measured an average execution speed of 20 268 exec/s. In comparison, WAFL achieves about 279 exec/s. Figure 8 shows the stack trace from Wemby’s bug report for this experiment. For reference, Figure 9 shows an excerpt of an identical report produced by clang ASan for the same bug. Wemby successfully detected the memory corruption vulnerability, pinpointed the vulnerable function, and accurately classified the issue as a stack-based buffer overflow.

8.2 From Memory Corruption to XSS with Wemby

We illustrate Wemby’s exploit facilitation capabilities using a memory corruption vulnerability in a Wasm module found on a website from our analysis (Section 4). This homeware and clothing retailer’s website enables customers to purchase items online and provides a feature for users to scan item barcodes for queries. After analyzing the target with Wemby, we discovered that this module invokes multiple calls to unsafe JavaScript functions. A noteworthy function is triggered when the user engages with the scanner: The Wasm module repeatedly uses `emscripten_run_script_string`, i.e., `eval()`, to check the current date using `Date.now().toString()`, and saves the result in linear memory.

Wemby identified that unsanitized data from a third-party WebSocket is being loaded into the Wasm memory. Furthermore, Wemby detected a memory error that allows an attacker to overwrite data in the Wasm module’s linear memory through this unsanitized data. All the data used by the unsafe JavaScript functions are located at static offsets in the linear memory and can be overwritten with this primitive.

The attack process is as follows: The attacker substitutes the network data with the exploit payload. When the victim visits the website, the payload overwrites the `Date.now().toString()` string at address 2294 in linear memory with the XSS payload. The vulnerability is triggered when the victim uses the barcode scanner to scan an item. Due to Web-Wasm’s stateful nature, the payload persists in the linear memory until the victim reloads the website. Every time the barcode scanner is reopened, the XSS payload is executed instead of the `Date.now().toString()` function.

This attack underscores the need to examine the interaction interface between a Wasm module and its environment. Wemby traces 460 functions, seven of which process unsanitized data and provide insight into data reaching 74 unsafe JavaScript API (Table 2) calls that ultimately facilitate the shown attack. Unlike orthogonal approaches [11, 36–38, 58], Wemby can uncover this bug and assist in crafting exploits.

8.3 Case Study: Zoom

We tested Wemby on Zoom due to its real-world relevance and complexity. Zoom's Wasm code processes audio and video frames, presenting a large attack vector. A memory corruption vulnerability could disrupt or hijack multiple clients' browsers. Zoom uses non-standard Wasm features like SIMD and bulk memory operations, and its multithreaded architecture challenges existing Wasm fuzzers [36, 58] and analysis tools [5, 11, 56]. Wemby successfully tackles these challenges, demonstrating its effectiveness in analyzing, reverse-engineering, and fuzzing complex websites.

Our analysis reveals several architectural security implications in Zoom's Wasm implementation. The core issue stems from Zoom's memory management: media frames are stored directly in linear memory (C-heap) for processing WebSocket payload data, while global objects for audio processing, video handling, and desktop sharing functionality are maintained at deterministic addresses. This predictable memory layout, combined with Wasm's inherent limitations in CFI and memory protection (see Section 3.2), exposes potential attack vectors. Specifically, these architectural choices make the application susceptible to COOP-style [77] and data-only [41, 42] attacks. Through Wemby analysis of Zoom's 18 765 functions, we identified six security-critical functions, with three processing unsanitized external data. Most notably, the `Video_Try_Analysis` function, responsible for video frame decoding, processes every incoming video frame and presents a significant security risk due to its continuous interaction with potentially malicious input.

The security implications are further amplified by Zoom's configuration-dependent behaviors and minimal input sanitization. Our assessment revealed that video data traversing Zoom servers undergoes limited sanitization for performance reasons, allowing potential attackers to inject up to 1424 bytes into the Wasm linear memory of all connected clients. This risk becomes particularly severe because Zoom's Wasm module maintains a consistent memory layout across all clients, allowing an attacker to reliably target the same memory addresses in every participant's browser. The predictable nature of these memory patterns, combined with the processing of unsanitized data, creates a particularly concerning attack surface that could potentially affect all connected clients in a conference call.

We detected a memory corruption bug with Wemby and responsibly disclosed it to Zoom. The bug is in the video encoding component: with a crafted video, the data used in the Wasm module crashes the application on the client side. The issue is a stack-based buffer overflow capable of overwriting the entire linear memory, and the execution halts only when the payload attempts to write beyond the linear memory's bounds. The payload fails to reach other clients because the application becomes unresponsive before it can initiate the process of sending it to other participants. In line with ethical guidelines, we refrain from injecting harmful data into Zoom servers to test our exploits, which could pose risks to other users. Thus, we confirm Wemby's ability to fuzz even highly complex web targets like Zoom.

9 Ablation Study

Due to the absence of standardized Web-Wasm benchmarks Web-Wasm, we analyzed 76 randomly selected websites from Section 4 that exhibit properties P1 to P2 and analyzed them with Wemby. In addition, we subjected these Wasm modules to fuzzing under two distinct configurations—one without the analysis provided by Wemby and another without Wemby's snapshot mechanism. This setup serves as an ablation study, demonstrating the influence of Wemby in identifying reproducible Wasm bugs, as well as the effect of fuzzing on the code coverage achieved by the fuzzer. Lastly, we compare the performance of Wemby with that of WAFL [36].

9.1 Bug Finding Capabilities

In this experiment, we measured the precision of Wemby's bug-finding capabilities. Wemby detected 2258 *crashes* in 510 functions across 17 modules. Multiple crashes can occur when the same bug

triggers different oracles, like a stack-based buffer overflow vulnerability that can manifest as either an *out-of-bounds linear memory write* or a *stack-cookie overwrite*. To address this, we *deduplicated* these crashes by comparing their coverage within the same function, indicating shared vulnerabilities. Overall, Wemby was able to detect 332 unique bugs.

Finding 1: Wemby’s analysis slashes false positives by a factor of six. Existing approaches to Wasm fuzzing lack crucial contextual information about the module’s original web embedding, potentially leading to false positives that would never manifest in real-world conditions. For instance, when fuzzing a Wasm function that expects a pointer to a valid string buffer, existing approaches might supply random integers as arguments. This causes the function to attempt memory access at arbitrary locations, triggering crashes that would never occur in the actual web context where only valid pointer values are passed. By faithfully incorporating the website’s state and context, Wemby identifies genuine bugs while eliminating such spurious crashes.

Our evaluation demonstrates this distinction: Wemby’s fuzzer identified 2258 crashes across 17 Wasm modules. Moreover, when we fuzzed the Wasm functions without the analysis discussed in Section 7.2—a method similar to existing approaches [36, 58]—the crash count escalated significantly, reaching 12 927 crashes, most of which are false positives. Furthermore, 29% of the Wasm modules could not be fuzzed without Wemby’s analysis because they led to program crashes due to the linear memory not being in a valid state, and pointer misclassifications caused immediate program termination. These findings underscore that Wemby’s website analysis is an *essential* requirement to fuzz Wasm modules from the Web.

Finding 2: Bugs found with Wemby are always reproducible. Our fuzzer reports bugs if either our ASan bug oracle is triggered or if it violates the Wasm runtime. However, as the browser lacks Wemby’s oracles, only Wasm runtime violations are directly visible in the browser. Thus, we focus on the 1208 crashes that violate the Wasm runtime and omit bugs found through our custom oracles for this analysis. For each crash, Wemby generates a reproducer HTML file that instantiates the Wasm module, loads the snapshot of the linear memory, and calls the vulnerable function with the crashing input from the fuzzer, replaying the bug in the browser. We can confirm that *every* Wasm sandbox violating bug found by Wemby is reproducible in this fashion.

9.2 Coverage

In this experiment, we evaluated the coverage improvement by comparing the code covered by Wemby with and without snapshots. While Fuzzm and WAFL require instrumentation of the Wasm module to receive feedback coverage, we can use AFL++’s default coverage metric, which counts the number of basic blocks covered by the fuzzer. This enables us to precisely measure the code coverage of Wemby without any instrumentation of the Wasm module.

Finding 3: Snapshots boost code coverage by 46% on average. Our results indicate that it is better to integrate the state of the linear memory during the execution of the Wasm function. Compared to existing fuzzers [36, 58], which lack such integration, Wemby’s approach covers, on average, 46% more code. This is because existing Wasm fuzzers cannot pass fuzzing barriers imposed by the state, for example, Figure 3 line 17. As the experiments indicate, Wemby overcomes these barriers, allowing for a holistic security analysis of Wasm modules on the Web.

9.3 Fuzzing Performance

We compare Wemby’s fuzzing performance to WAFL [36], as Fuzzm [58] is not included in our evaluation due to compatibility issues¹. WAFL and Fuzzm are fuzzers designed for standalone Wasm

¹See <https://github.com/fuzzm/fuzzm-project/issues/5>.

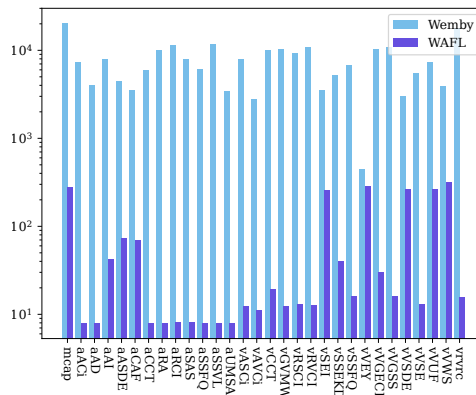


Fig. 10. Executions per second per core achieved by WAFL and Wemby. The X-Axis abbreviates the function names of the Zoom Wasm module.

applications utilizing WASI that require targets to have a single entry point. Web applications neither utilize WASI nor define a main method, rendering these fuzzers incompatible. To nevertheless enable a comparison, we generate synthetic main methods. Here, we leverage our web application analysis (see Section 7.2) to identify relevant entry functions. We then designate these functions as the main entry point for the other fuzzers. We then modify the targets to include main methods that read input from stdin and invoke one of the identified functions with the input as an argument, making them compatible with WAFL.

Finding 4: Wemby significantly outperforms WAFL. Figure 10 shows the execution speed per core achieved by Wemby and WAFL, respectively, running on the motivating example (mcap) and functions from Zoom’s Wasm modules. On average Wemby’s fuzzer is 232.60 times faster than WAFL. Wemby has about three orders of magnitude more executions per second, comparable to native fuzzing solutions. This speed-up results from our design choice of compiling the Wasm module to native code (cf. Section 7.3) instead of instrumenting the VM. Additionally, after reviewing these results, we found that the number of false alarms (see Finding 1) further impedes WAFL’s performance because each crash results in a restart of the Wasm VM, slowing down WAFL. Since Wemby omits a Wasm VM, it does not suffer from this issue. Hence, besides being unable to fuzz complex targets, WAFL’s low speed further limits practical vulnerability analysis.

10 Related Work

Large-Scale Web Studies. Assessing the security of the Web at large is mainly done via crawling studies. These cover a wide range of security-related aspects, such as the prevalence of reflected [7, 60, 64, 90] and stored Client-Side XSS [85], employed protection mechanisms [52], DOM clobbering [49], Client-Side Cross Site Request Forgery [48, 50] or prototype pollution [47]. Additionally, more general aspects of the web have been studied, such as compliance to the HTML specification [35], JavaScript library usage [70], and how third-party JavaScript interacts hinders security mechanisms [84].

(Ab)use of WebAssembly. Among Wasm’s first uses and most widely studied aspects was its role in cryptojacking [40, 53, 68]. Popularized in 2018, website operators implemented cryptocurrency mining functionality in Wasm and used the computing resources of unsuspecting visitors. Similarly, Wasm has been shown to be able to obfuscate JavaScript malware by Romano et al. [75]. More generally, the first study on its prevalence on the web at large was conducted by Musch et al. [67]. Wasm has since found adoption in native applications as well, for example, to sandbox untrusted code [9]. Similarly, Firefox also employs WebAssembly to sandbox untrusted third-party libraries [69].

Wasm Binary Analysis. Previous work on assessing the security of Wasm code focuses on WASI applications. Fu et al. [27] implement taint tracking, Stiévenart et al. [86] and Brito et al. [11] develop static analysis frameworks. More generally, SnowWhite [55], WasPur [76] and WasmRev [43] attempt to recover high-level types to aid further analysis, and Lehmann et al. [57] show how aspects of Wasm make call-graph construction difficult. Eunomia [38] and SeeWasm [37] are symbolic execution engines. Wasabi [56], Walrus [93], and Wasm-R3 [5] are dynamic analysis frameworks that can instrument Wasm code to perform diverse execution analyses. WAFL [36] and Fuzzm [58] fuzz Wasm code but are unsuitable for browser targets as they ignore the website environment. In this paper, we address these browser-specific challenges.

Wasm Hardening and Runtime Reliability. Likewise, current research on hardening Wasm also focuses on WASI. PKUWA [59] uses Intel MPK to split the Wasm linear memory into *domains*, confining memory errors into function-scoped memory areas. Further, there is research that analyzes not the Wasm code itself but the security of the sandbox, e.g., using formal verification [45, 46], or re-compilation using Rust’s verification [9]. Fuzzing the Wasm runtime, as demonstrated by WASMaker [14], Wapplique [98], and WADIFF [99], is a promising research direction to enhance the Wasm SFI further. While this enhances runtime robustness, it cannot detect memory corruption within Wasm binaries. Ultimately, these errors can evade SFI, as demonstrated by this work.

Library Fuzzing. Fuzzing is a popular technique to assess the security of software [8] and hardware [62] and to find impactful vulnerabilities. Grammar-based input fuzzers [23, 30, 34, 96] infer invariants in well-structured data to produce impactful test cases. Recent research has focused on fuzzing targets on different platforms [18, 44, 61, 73, 82], which share properties with the attack surface of Wasm modules. In this paper, we use AFL++ [25] as a fuzzer and leave more sophisticated fuzzing approaches for future work.

11 Conclusion and Summary

In conclusion, our research reveals a significant security risk associated with the widespread trust in data interfacing with Wasm modules, a practice adopted by over 29 411 websites. This trust, when combined with memory corruption errors, can lead to remote code execution, presenting a new threat model. We detected 2782 domains vulnerable to this threat model. To address this risk, we present Wemby, the first and efficient approach for analyzing WebAssembly-powered websites. Wemby outperforms existing solutions in detecting remotely exposed memory corruption errors. Its advanced tracing and taint-tracking capabilities make it a valuable tool for reverse-engineering complex, closed-source targets. Wemby revealed several security-critical functions in websites, including memory corruption bugs, demonstrating its effectiveness. This work emphasizes the need for robust security measures in WebAssembly-powered websites and presents a superior solution with Wemby.

Acknowledgments

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through SFB 1119 – 236615297, project S2, and under Germany’s Excellence Strategy – EXC 2092 CASA – 390781972.

References

- [1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "CFI: Principles, implementations, and applications". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2005.
- [2] Zubair Ahmad, Stefano Calzavara, Samuele Casarin, and Ben Stock. "Information flow control for comparative privacy analyses". In: *International journal of information security* (2024).
- [3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing symbolic execution with veritesting". In: *Proc. of the International Conference on Software Engineering (ICSE)*. DOI: [10.1145/2568225.2568293](https://doi.org/10.1145/2568225.2568293).
- [4] Amazon AWS. *Amazon Interactive Video Service (IVS)*. <https://aws.amazon.com/ivs/>. Acc.: 2024-09-01. 2024.
- [5] Doehyun Baek, Jakob Getz, Yuseung Sim, Daniel Lehmann, Ben Titzer, Sukyoung Ryu, and Michael Pradel. "Wasm-R3: Record-Reduce-Replay for Realistic and Standalone WebAssembly Benchmarks". In: *Proceedings of the ACM on Programming Languages: Object-Oriented Programming, Systems, Languages & Applications*. 2024.
- [6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51 (2018).
- [7] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. "Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis". In: *Proc. of the European Workshop on System Security (EUROSEC)*. 2021.
- [8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2016.
- [9] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. "Provably-Safe Multilingual Software Sandboxing using WebAssembly". In: *USENIX Security Symposium*. 2022.
- [10] Soumaya Boussaha, Lukas Hock, Miguel Bermejo, Ruben Cuevas Rumin, Angel Cuevas Rumin, David Klein, Martin Johns, Luca Compagna, Daniele Antonioli, and Thomas Barber. "FP-tracer: Fine-grained Browser Fingerprinting Detection via Taint-tracking and Multi-level Entropy-based Thresholds". In: *Privacy Enhancing Technologies Symposium (PETS)*. 2024. DOI: [10.56553/popets-2024-0092](https://doi.org/10.56553/popets-2024-0092).
- [11] Tiago Brito, Pedro Lopes, Nuno Santos, and José Frago Santos. "Wasmati: An efficient static vulnerability scanner for WebAssembly". In: *Comput. Secur.* 118 (July 2022).
- [12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *OSDI*. 2008.
- [13] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. "Symbolic execution for software testing in practice: preliminary assessment". In: *Proc. of the International Conference on Software Engineering (ICSE)*. DOI: [10.1145/1985793.1985995](https://doi.org/10.1145/1985793.1985995).
- [14] Shangdong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. "WASMaker: Differential testing of WebAssembly runtimes via semantic-aware binary generation". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024.
- [15] Orçun Çetin, Mohammad Hanif Jhaveri, Carlos Gañán, Michel van Eeten, and Tyler Moore. "Understanding the role of sender reputation in abuse reporting and cleanup". In: *J Cyber Secur* 2.1 (Dec. 2016), pp. 83–98. ISSN: 2057-2085. DOI: [10.1093/cybsec/tyw005](https://doi.org/10.1093/cybsec/tyw005).
- [16] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. *Unleashing Mayhem on Binary Code*. 2012.
- [17] *Chrome Platform Status*. <https://chromestatus.com/metrics/feature/timeline/popularity/2237>. Acc.: 2024-2-22. 2024.
- [18] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. "SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing". In: *USENIX Security Symposium*. 2022.
- [19] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *USENIX Security Symposium*. 1998.
- [20] D Crocker. *Mailbox names for common services, roles and functions*. Tech. rep. May 1997. DOI: [10.17487/rfc2142](https://doi.org/10.17487/rfc2142).
- [21] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection". In: *USENIX Security Symposium*. 2014.
- [22] Emscripten. *Calling JavaScript from C/C++*. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#calling-javascript-from-c-c. Acc.: 2024-09-01. 2024.
- [23] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. "Fuzzing JavaScript interpreters with coverage-guided reinforcement learning for LLM-based mutation". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*.
- [24] Fingerprint. *FingerprintJS: Browser fingerprinting library*. 2024.
- [25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2020.
- [26] Jonathan Foote. *Hijacking the control flow of a WebAssembly program*. <https://www.fastly.com/blog/hijacking-control-flow-webassembly>. Acc.: 2024-09-01. 2018.

- [27] William Fu, Raymond Lin, and Daniel Inge. “TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly”. In: (2018). arXiv: [1802.01050 \[cs. CR\]](https://arxiv.org/abs/1802.01050).
- [28] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. “Out of Control: Overcoming Control-Flow Integrity”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2014.
- [29] Google. *skia: Skia is a complete 2D graphic library for drawing Text, Geometries, and Images*. Acce.: 2024-09-01. 2024.
- [30] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. “FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities”. In: *Network and Distributed System Security Symposium (NDSS)*. 2023.
- [31] WebAssembly Community Group. *wabt: The WebAssembly Binary Toolkit*. 2023.
- [32] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. “SpecuSym: Speculative symbolic execution for cache timing leak detection”. In: *Proc. of the International Conference on Software Engineering (ICSE)*. 2020.
- [33] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J F Bastien. “Bringing the web up to speed with WebAssembly”. In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017.
- [34] Hyungseok Han, Donghyeon Oh, and Sang Kil Cha. “CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines”. In: *Network and Distributed System Security Symposium (NDSS)*. 2019.
- [35] Florian Hantke and Ben Stock. “HTML Violations and Where to Find Them: A Longitudinal Analysis of Specification Violations in HTML”. In: *Internet Measurement Conference (IMC)*. 2022.
- [36] Keno Haßler and Dominik Maier. “WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots”. In: *Reversing and Offensive-oriented Trends Symposium*. 2021.
- [37] Ningyu He, Zhehao Zhao, Hanqin Guan, Jikai Wang, Shuo Peng, Ding Li, Haoyu Wang, Xiangqun Chen, and Yao Guo. “SeeWasm: An efficient and fully-functional symbolic execution engine for WebAssembly binaries”. In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024. doi: [10.1145/3650212.3685300](https://doi.org/10.1145/3650212.3685300).
- [38] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. “Eunomia: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries”. In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023. doi: [10.1145/3597926.3598064](https://doi.org/10.1145/3597926.3598064).
- [39] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. “An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases”. In: *The Web Conference*. 2021.
- [40] Geng Hong, Zheming Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Hai-Xin Duan. “How You Get Shot in the Back: A Systematical Study about Cryptojacking in the Real World.” In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2018. doi: [10.1145/3243734.3243840](https://doi.org/10.1145/3243734.3243840).
- [41] Hong Hu, Zheng Leong Chua, Sendriou Adrian, Prateek Saxena, and Zhenkai Liang. “Automatic generation of data-oriented exploits”. In: *USENIX Security Symposium*. 2015.
- [42] Hong Hu, Shweta Shinde, Sendriou Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2016. doi: [10.1109/SP.2016.62](https://doi.org/10.1109/SP.2016.62).
- [43] Hanxian Huang and Jishen Zhao. “Multi-modal Learning for WebAssembly Reverse Engineering”. In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023.
- [44] Bo Jiang, Ye Liu, and W K Chan. “ContractFuzzer: fuzzing smart contracts for vulnerability detection”. In: *Proc. of the International Conference on Automated Software Engineering (ASE)*.
- [45] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. “WaVe: a verifiably secure WebAssembly sandboxing runtime”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2023.
- [46] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. “Доверяй, но проверяй: SFI safety for native-compiled Wasm”. In: *Proceedings 2021 Network and Distributed System Security Symposium*. 2021.
- [47] Zifeng Kang, Song Li, and Yinzhi Cao. “Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites.” In: *Network and Distributed System Security Symposium (NDSS)*. 2022.
- [48] S. Khodayari, T. Barber, and G. Pellegrino. “The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2024. doi: [10.1109/SP54263.2024.00098](https://doi.org/10.1109/SP54263.2024.00098).
- [49] Soheil Khodayari and Giancarlo Pellegrino. “It’s (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2023.
- [50] Soheil Khodayari and Giancarlo Pellegrino. “JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals.” In: *USENIX Security Symposium*. 2021.

- [51] Robin Kirchner, Jonas Möller, Marius Musch, David Klein, Konrad Rieck, and Martin Johns. "Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind Cross-Site Scripting". In: *USENIX Security Symposium*. 2024.
- [52] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. "Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2022.
- [53] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense." In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2018. doi: [10.1145/3243734.3243858](https://doi.org/10.1145/3243734.3243858).
- [54] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything Old is New Again: Binary Security of Web-Assembly". In: *USENIX Security Symposium*. 2020.
- [55] Daniel Lehmann and Michael Pradel. "Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries". In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2022. doi: [10.1145/3519939.3523449](https://doi.org/10.1145/3519939.3523449).
- [56] Daniel Lehmann and Michael Pradel. "Wasabi: A framework for dynamically analyzing webassembly". In: *Proceedings of the Twenty-Fourth International* (2019).
- [57] Daniel Lehmann, Michelle Thalakkottur, Frank Tip, and Michael Pradel. "That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly". In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023. doi: [10.1145/3597926.3598104](https://doi.org/10.1145/3597926.3598104).
- [58] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. "Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly". In: (Oct. 2021). arXiv: [2110.15433](https://arxiv.org/abs/2110.15433) [cs.CR].
- [59] Hanwen Lei, Ziqi Zhang, Shaokun Zhang, Peng Jiang, Zhineng Zhong, Ningyu He, Ding Li, Yao Guo, and Xiangqun Chen. "Put your memory in order: Efficient domain-based memory isolation for WASM applications". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2023. doi: [10.1145/3576915.3623205](https://doi.org/10.1145/3576915.3623205).
- [60] Sebastian Lekies, Ben Stock, and Martin Johns. "25 Million Flows Later: Large-scale Detection of DOM-based XSS". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2013.
- [61] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. "VD-guard: DMA guided fuzzing for hypervisor virtual device". In: *Proc. of the International Conference on Automated Software Engineering (ASE)*.
- [62] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE: baseband sanitized fuzzing through emulation". In: *Proc. of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2020.
- [63] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. "Security chasms of wasm". In: *NCC Group Whitepaper* (2018).
- [64] William Melicher, Anupam Das, Mahmood Sharif, Lujio Bauer, and Limin Jia. "Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting". In: *Network and Distributed System Security Symposium (NDSS)*. 2018.
- [65] WebAssembly CG members. *WebAssembly Security*. <https://webassembly.org/docs/security/>. Acc.: 2024-09-01. 2024.
- [66] Microsoft. *Control Flow Guard*. <https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard>.
- [67] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "New kid on the web: A study on the prevalence of WebAssembly in the wild". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2019.
- [68] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. "Thieves in the browser: Web-based crypto-jacking in the wild". In: *Proc. of the International Conference on Availability, Reliability and Security (ARES)*. 2019.
- [69] Shravan Narayan, Craig Disselkoe, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting fine grain isolation in the Firefox renderer". In: *USENIX Security Symposium*. 2020.
- [70] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. "You are what you include: large-scale evaluation of remote javascript inclusions." In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2012. doi: [10.1145/2382196.2382274](https://doi.org/10.1145/2382196.2382274).
- [71] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* 7.49 (1996), pp. 14–16.
- [72] Jannis Rautenstrauch, Metodi Mitkov, Thomas Helbrecht, Lorenz Hetterich, and Ben Stock. "To Auth or Not To Auth? A Comparative Analysis of the Pre- and Post-Login Security Landscape". In: *Proc. of the IEEE Symposium on Security and Privacy*. 2024. doi: [10.1109/SP54263.2024.00094](https://doi.org/10.1109/SP54263.2024.00094).
- [73] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. "EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2023.
- [74] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z Snow, and Michalis Polychronakis. "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses". In: *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017.

- [75] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. “Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly.” In: *Proc. of the IEEE Symposium on Security and Privacy*. 2022. DOI: [10.1109/SP46214.2022.9833626](https://doi.org/10.1109/SP46214.2022.9833626).
- [76] Alan Romano and Weihang Wang. “Automated WebAssembly Function Purpose Identification With Semantics-Aware Analysis”. In: *The Web Conference*.
- [77] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. “Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2015. DOI: [10.1109/SP.2015.51](https://doi.org/10.1109/SP.2015.51).
- [78] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *Proc. of the IEEE Symposium on Security and Privacy*. DOI: [10.1109/SP.2010.26](https://doi.org/10.1109/SP.2010.26).
- [79] R. Sekar. “An Efficient Black-box Technique for Defeating Web Application Attacks.” In: *Network and Distributed System Security Symposium (NDSS)*. 2009.
- [80] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: a fast address sanity checker”. In: *Proceedings of the 2012 USENIX conference on Annual Technical Conference (ATC’12)*. 2012.
- [81] Hovav Shacham et al. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”. In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2007.
- [82] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. “Fuzz on the Beach: Fuzzing Solana Smart Contracts”. In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2023.
- [83] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2013. DOI: [10.1109/SP.2013.45](https://doi.org/10.1109/SP.2013.45).
- [84] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. “Who’s Hosting the Block Party? Studying Third-Party Blockage of CSP and SRL.” In: *Network and Distributed System Security Symposium (NDSS)*. 2021.
- [85] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. “Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.” In: *Network and Distributed System Security Symposium (NDSS)*. 2019.
- [86] Quentin Stiévenart and Coen De Roover. “Compositional Information Flow Analysis for WebAssembly Programs”. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020.
- [87] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. “How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security”. In: *USENIX Security Symposium*. 2017.
- [88] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. “Didn’t you hear me? - towards more successful web vulnerability notifications”. In: *Network and Distributed System Security Symposium (NDSS)*. 2018. DOI: [10.14722/ndss.2018.23171](https://doi.org/10.14722/ndss.2018.23171).
- [89] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. “Hey, you have a problem: On the feasibility of Large-Scale web vulnerability notification”. In: *USENIX Security Symposium*. 2016.
- [90] Ben Stock, Stephan Pfister, Bernd Kaiser, Sebastian Lekies, and Martin Johns. “From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting”. In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2015.
- [91] PaX Team. “PaX Address Space Layout Randomization (ASLR)”. In: <https://pax.grsecurity.net/docs> (2001).
- [92] The Clang Team. *DataFlowSanitizer*. <https://clang.llvm.org/docs/DataFlowSanitizer.html>. Accessed: 2024-09-01. 2023.
- [93] *Walrus: A WebAssembly transformation library*. <https://github.com/rustwasm/walrus>.
- [94] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. “TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection”. In: *Proc. of the IEEE Symposium on Security and Privacy*. 2010.
- [95] WebAssembly System Interface Subgroup Charter. WASI. <https://wasi.dev/>. Acc.: 2024-09-01. 2024.
- [96] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. “Automated conformance testing for JavaScript engines via deep compiler fuzzing”. In: *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
- [97] Eric Zeng, Frank Li, Emily Stark, A Felt, and Parisa Tabriz. “Fixing HTTPS Misconfigurations at Scale: An Experiment with Security Notifications”. In: (2019).
- [98] Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. “Waplique: Testing WebAssembly Runtime via Execution Context-Aware Bytecode Mutation”. In: *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024.
- [99] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. “WADIFF: A Differential Testing Framework for WebAssembly Runtimes”. In: *Proc. of the International Conference on Automated Software Engineering (ASE)*. 2023.