2026 IEEE 11th European Symposium on Security and Privacy (EuroS&P) | 979-8-3195-1074-7/26/$31.00 ©2026 IEEE | DOI: 10.1109/EuroSP68448.2026.00077

SandVenture: Escaping JavaScript Sandboxes with Objective-driven Input Generation

Mingi Jung
UNIST

Hyeon Heo
ENKI WhiteHat

Seongil Wi
UNIST

***Abstract*—JavaScript (JS) sandboxing plays a crucial role in ensuring the integrity of web services by establishing safe execution environments for untrusted JS code. While numerous sandbox frameworks have been proposed, few studies have systematically explored methods for identifying sandbox-escaping bugs. Prior work developed automated systems to find such bugs, but their reliance on regression test suites inherently restricts comprehensive exploration of complex attack vectors.**

We present SandVenture, a JS sandbox fuzzer that generates diverse inputs guided by concrete objectives designed to break the security guarantees enforced by the sandbox under test. We introduce novel and systematic ways of generating and mutating JS inputs driven by two objectives: (1) referencing a comprehensive set of accessible built-in objects in diverse contexts and (2) exploiting interface bridges between the sandbox host and guest code. SandVenture successfully discovered 85 sandbox-escaping bugs, including four CVEs, in 11 sandboxes. Compared to prior work, SandVenture found 31% more bugs, demonstrating the efficacy of objective-driven input generation in vulnerability discovery.

***Index Terms*—JS sandbox escaping, JS sandbox fuzzing, objective-driven input generation**

1. Introduction

Modern web applications make use of cross-origin resources, including JavaScript (JS) snippets from third parties to implement content-rich and responsive web services. A total of 88.45% of the Alexa top 10,000 sites include at least one third-party script [1], and 85% of the top one million sites include an average of 19 third-party JS snippets from various domains [2]. Moreover, an increasing number of server-side Node.js applications support the execution of user-provided JS snippets. For instance, platform as a service (PaaS) [3], [4] provides an execution environment where users run arbitrary JS code in the Node.js cloud.

Unfortunately, this growing trend of deploying user-provided or third-party JS snippets exacerbates security risks, particularly regarding the integrity of these scripts; the adversary is able to provide malicious scripts that harvest and exploit sensitive first-party resources. This is possible because these scripts run from the same origin as the first-party context, granting them full privileges to access sensitive resources [1], [5], [6], [7].

To mitigate the potential security risks that untrusted JS code poses, security developers and researchers have introduced the concept of JS sandboxing. JS sandboxing involves isolating the execution of untrusted JS code from the execution environment of trusted JS code. Specifically, JS sandboxing involves host and guest code: (1) *host code* refers to a trusted JS snippet that establishes the sandboxing environment, and (2) *guest code* refers to an untrusted JS snippet that needs to be sandboxed.

Numerous studies and existing JS sandbox frameworks have proposed diverse ways of isolating the execution environment for guest code using various isolation primitives [8], [9], [10], [11], [12]. They have been deployed for various purposes such as JS playground [13], [14], page rendering [15], application development [16], an online game [17], custom crawler [18], and malware analysis [19]. Three major sandbox frameworks, SES, vm2, and isolated-vm, are downloaded 500K times per week, which demonstrates the high demand for JS sandboxing [20], [21], [22].

JS sandbox frameworks are designed to enforce a security property in which the execution of guest code should not result in unauthorized access to host resources. When a sandbox policy permits access to a limited set of host resources, the guest code must be strictly confined to accessing and manipulating only those that are explicitly allowed.

However, numerous sandbox-escaping bugs have been reported, each of which enables guest code to break the security promise of its underlying sandbox [23], [24], [25]. The systematic identification of such bugs is imperative for evaluating and improving the security of sandbox frameworks.

Recently, AlHamdan and Staicu [26] presented SandDriller, the first approach to test JS sandbox frameworks. SandDriller leverages existing regression JS tests as a seed set. For each seed file, it attempts prototype pollution on all references (e.g., variables) as an oracle check to detect any foreign references. However, this approach is inherently limited by its reliance on the built-in objects present in the original seed files. As a result, it cannot explore unseen or deeply nested references, leading to a restricted search space.

In this paper, we propose a novel input generation technique guided by concrete *objectives*. These objectives are designed to generate JS inputs with explicit intent to escape sandbox restrictions. In particular, our approach combines both generational and mutational strategies to prepare a more comprehensive set of JS inputs that target various corner cases of sandbox enforcement logic.

To define concrete objectives for sandbox testing, we asked the following questions: Are there any similar pat-

DOI 10.1109/EuroSP68448.2026.00077

terns in sandbox-breaking JS exploits? If so, how can one leverage those patterns to systematically generate JS test inputs breaking JS sandbox frameworks? To answer these questions, we conducted a qualitative analysis of 67 known vulnerabilities and their corresponding proof-of-concept (PoC) exploits.

We observed that 74% of PoC exploits share one pattern in which *a sandbox-breaking exploit often references a built-in object whose prototype chain holds the references to host resources*. The adversary obtains a built-in object reference and then attempts to access host resources by exhaustively referencing all the reference paths that lie in a prototype chain stemming from this built-in object reference. Also, we found that 26% of PoC exploits exhibited another new pattern: *a sandbox-breaking exploit often leverages a host-guest bridge to inject elements that embed JS code or invoke the built-in functions allowing for JS execution*.

To address the first observation, we define the first objective in which the test inputs should exploit references to a comprehensive set of built-in objects accessible from guest code and then perform prototype pollution on each of those built-in object references. For the second observation, we devise another new objective: test inputs should attempt to inject all HTML elements that can embed JS code and invoke all built-in functions that execute JS code. To this end, we reference the ECMAScript standard, known Cross-Site Scripting (XSS) payloads, and XSS cheat sheets to cover all known ways for embedding JS code via HTML elements and invoking functions capable of executing JS code.

To demonstrate a proof of concept that implements these generation strategies, in this paper, we propose SandVenture, a fuzzing framework designed to find sandbox-escaping bugs. Given a JS sandbox framework and seed corpus, SandVenture generates JS tests, each of which abuses guest-accessible built-in objects or host-guest interfaces to inject JS code into the host execution context according to the aforementioned two objectives. SandVenture employs both generational and mutational fuzzing: it first generates diverse JS inputs based on six input generation strategies, then mutates these inputs with the seed corpus to maximize the chances of uncovering sandbox-escaping bugs.

We evaluated SandVenture on 11 sandbox frameworks. We performed six testing campaigns; all rounds lasted 24 hours in total. SandVenture found 85 previously unreported bugs, including four CVEs, each of which breaks the security property the sandbox was intended to enforce. Our experiment also shows that SandVenture discovers 31% more unique bugs than SandDriller under the same time and seed input conditions.

In summary, this paper makes the following contributions:

1. We conduct a systematic study on 67 known sandbox-escaping bugs, leading to two observations about common patterns in JS inputs.
2. We propose a novel input generation algorithm that (a) references diverse built-in objects and (b) abuses guest-host bridges, leading to the discovery of 85 previously unreported bugs, including four CVEs. Eleven of them have been patched, with SES awarding a bounty of 5,000 USD.
3. We design and implement SandVenture, a JS sandbox fuzzing tool based on these objectives. Our tool is available in our repository (§F).

2. Background

2.1. JavaScript Sandboxing

JS sandboxing is a technique that establishes an isolated execution environment to run JS code snippets [8], [9], [10], [11], [12]. A first-party website confines the execution behaviors of untrusted advertising JS code via sandboxing [9], [11]. For example, MetaMask (a Web3 crypto wallet reporting 30M monthly users in 2025 [27]) runs third-party plugin programs in an SES sandbox [28] to isolate them from the wallet application. Furthermore, a server-side web application often uses JS sandboxing to run user-provided JS code to compute the execution result. For instance, isolated-vm [29] is used to isolate user-provided JS code in an online game [17] and a search service provider [18].

Existing sandboxing techniques are categorized into runtime-based and language-based isolation methods [9], [30], [31], [32]. Runtime-based isolation leverages system primitives that enable the isolation of an execution environment, such as process [30], web worker [9], [31], or origin-separated iframe [32], [33]. In contrast, language-based isolation implements lightweight confinement mechanisms, creating an isolated JS scope with a set of accessible but limited built-in objects.

JS sandboxing frameworks can also be categorized into server-side and client-side frameworks based on their deployment side. Server-side sandboxes confine given JS code within server-side Node.js applications, while client-side frameworks create a sandboxing environment in browsers in which the HTML content is rendered. Existing sandboxing frameworks support either or both deployment sides; vm2 [34] is a server-side sandboxing framework for Node.js applications, while SES [28] supports both sides to confine arbitrary JS code.

Notation. In a JS sandboxing framework, the first party that establishes a sandboxing environment is the *host* party. A trusted JS snippet from this host party is the *host* code, while an untrusted JS snippet that should be run within a sandboxing environment is the *guest* code. The host party sets a *sandboxing policy* that defines read/write privileges for APIs and built-in objects accessible from the guest code. This policy can be represented as a list of key-value pairs, where the key indicates a built-in function or an object accessible from the guest code, and the value describes the read/write privilege for the built-in function or object that the corresponding key indicates.

Most sandboxing frameworks are designed to establish an isolated scope for the guest code, explicitly preventing it from accessing the host code. A *scope* refers to an execution context in which values and expressions are visible or can be referenced [35]. In the context of JS sandboxing, the built-in objects and functions available to guest code may vary depending on the scope. For instance, the default scope in vm2, a server-side sandbox, exposes a different set of built-in objects and functions than TreeHouse, which is a client-side sandbox implemented as a

```
1 const {VM} = require("vm2");
2 let vmInstance = new VM();
3 const guest_code = `
4     let proc = require('child_process');
5     proc.execSync('[arbitrary shell command]'); `
6 vmInstance.run(guest_code)
7 // Error: Unable to access require() from guest scope
```

Figure 1: Example of sandboxing JS code using vm2.

```
1 <html>
2   <script src="treehouse/lib/require.js"></script>
3 // Error: Unable to access document from guest scope
4   <script type="text/x-treehouse-javascript"
5     treehouse-sandbox-name="worker1"
6     src="third-party.js"
7   ></script>
8 </html>
```

(a) Host code.

```
1 document.cookie =
2     "flag=flag"
```

(b) Guest code (`third-party.js`).

Figure 2: Example of sandboxing JS code using TreeHouse.

research prototype [9]. The term *built-in object* refers to a JS object accessible from the global scope [36]. This definition also covers JS objects that can be accessible by iteratively referencing the `global` object and its predecessors or ancestors, composing a reference path (e.g., `global.Proxy` and `global.Object.name`).

Sandbox objective. AlHamdan and Staicu summarized four security objectives (i.e., preventing side effects, restricting privileged operations, blocking the main loop, and preventing crashes) that existing sandboxes are designed to attain [26]. However, the most important security property is that *the guest code should be unable to access (read or write) host resources that are not allowed by a given sandboxing policy.*

Figure 1 shows a JS snippet using vm2, a server-side sandboxing library. The host code in Lines 1–3 prepares a sandboxing environment in which the guest code in Lines 4–5 runs on Line 6. Figure 2 also shows the isolation of the guest code in the host HTML instance (Lines 4–7 in Figure 2a) using TreeHouse, a client-side sandbox. The host code specifies no policy, thus allowing no access to any host resources by default. Therefore, accessing the `require` function or `document` object from guest code that exists in the host scope (e.g., Line 4 of Figure 1 or Line 1 of Figure 2b) should be blocked by the underlying sandbox.

2.2. Sandbox-escaping Bug

A bug in a JS sandbox poses a serious security threat. Consider a site operator who deploys a sandbox framework that forbids access to host objects by a given sandboxing policy. A sandbox-escaping bug may allow the adversary to access host objects, which obsoletes the need to deploy sandbox frameworks. For example, an adversary (i.e., a malicious third-party JS provider) is able to execute arbitrary commands on the server side using the `require` function, leading to remote code execution, or manipulate document cookies on the browser side by accessing the `document.cookie` object.

The guest code in Figure 3 triggers a vulnerability in vm2 that allows the guest code to access host resources. The JS code in Line 1 obtains a `Promise` built-in object by calling the `import` function. Note that this object is referenced as a host object because vm2 did not properly

```
1 let ret = import("XX");
2 ret.constructor.constructor('return process')().main
3   Module.require('child_process').execSync('[command]');
```

Figure 3: Guest code triggering a sandbox-escaping bug (CVE-2021-23449) in vm2.

```
1  const {VM} = require("vm2");
2  let vmInstance = new VM();
3  const guest_code = `
4    try {
5      function recursive() {
6        Error().stack;
7        recursive();
8      };
9      recursive();
10   } catch (referenced_obj) {
11     referenced_obj.constructor.constructor('return
12         process')().mainModule.require('child_process').
13         execSync('[arbitrary shell command]');} `
14 vmInstance.run(guest_code)
```

Figure 4: Sandbox-escaping vulnerability (CVE-2021-23555) in vm2 reflecting observation 1.

wrap it as a guest object. Lines 2–3 then access its root prototype, which returns the `process` object in the host execution environment. This immediately violates the aforementioned security objective, breaking the security guarantee of JS sandboxing, and allowing the adversary to execute arbitrary commands.

Prototype chain. Following the ECMAScript standard, every JS object has its own prototype chain that starts from its prototype object to the root prototype (i.e., `Object.prototype`) [37]. There are traditionally two prototype pollution patterns: one through `__proto__` and the other through `constructor` [38], [39], [40], [41], [42]. Given an object instance, accessing the `__proto__` property gives a reference to its prototype object, and traversing the prototype chain of this given object enables us to reference all prototype ancestors. On the other hand, given an object instance, accessing the `constructor` property returns a reference to the constructor function that creates the instance object [43]. Also, the `__proto__` property of this constructor returns the prototype of the given object instance. Therefore, the combinations of the prototype and constructor references enable the exploration of diverse JS objects.

3. Motivation

3.1. Preliminary Study

Is there a common pattern in JS exploits that triggers sandbox-escaping bugs? How can we leverage such a pattern to generate JS inputs that lead to the discovery of sandbox-escaping bugs? To answer these questions, we conducted a qualitative study and analyzed 67 sandbox bugs, including 18 CVEs, that break the security promise held by various sandbox frameworks. As a result, we derived two key observations about sandbox-escaping bugs.

We argue that our two observations are general enough, since we have conducted a best-effort collection and analysis of all sandbox-escaping bugs reported between 2009 and 2025. In Appendix, we present details of the studied 67 vulnerabilities, along with their patterns (§A).

Observation 1. We found that 49 out of the 67 sandbox bugs (74%) were caused by allowing guest code to obtain

```
1 let div = document.createElement("div")
2 div.style = "background-image:
3             url(javascript:document.cookie='flag=flag')"
4 document.body.appendChild(div)
```

Figure 5: Sandbox-escaping vulnerability (SecurityAdvisory20120502) in Caja reflecting observation 2 (guest code).

built-in object references of which the prototype chains contain a reference path to host resources, thus escaping the sandbox confinement. To trigger these 49 bugs, the guest code typically follows a two-step process. It starts by obtaining a reference to a built-in object by accessing it in the guest scope, invoking built-in functions (e.g., `import()` in Figure 3), or referencing error objects triggered by exceptions. Then, it traverses the prototype chain of the obtained object to access host-level resources.

Figure 4 shows a POC exploit exhibiting this common pattern. In particular, while calling the `recursive` function defined in Line 5 recursively through Line 7, the calling contexts are accumulated on the call stack, and the contents of the stack are continuously returned through Line 6. In the end, the size of the stack continues to increase, resulting in an error object, `RangeError` (`referenced_obj` in Line 10), which allows access to the scope of the host code.

While specific built-in objects that are exploited may vary depending on the target CVEs and sandbox frameworks, we observed that there is a common pattern of leveraging accessible built-in objects. Other PoC exploits may reference other built-in objects that could hold a path to the host scope. We further categorize such JS objects in §4.

> *Observation 1.* Sandbox-breaking often involves obtaining a built-in object reference of which the prototype chain holds a reference leading to the scope of host code.

Observation 2. We conducted further analysis and found that 18 out of the 67 bugs (26%) involve guest code executing arbitrary JS code in the host scope by exploiting the interfaces allowed by a given sandbox policy. This is typically achieved by injecting HTML elements that contain JS code into the host environment via host-guest bridges. The sandbox frameworks should have prevented the injection of such HTML elements, but due to shortcomings in checking the various methods of embedding JS code into HTML elements, the sandbox objective is compromised. In addition, we observed that a permissive sandbox policy allowing the invocation of built-in functions enables the guest code to execute arbitrary JS code in the host scope.

Figure 5 shows a PoC exploit targeting a client-side sandbox. Google Caja [44], an archived client sandbox, had a policy that allows guest code to write URLs to a `style` attribute. An attacker can abuse this policy to execute arbitrary JS code in the host context by redirecting to a `javascript:` URL through this attribute (Line 3).

> *Observation 2.* Client-side sandbox breaking often involves exploiting a host-guest bridge that enables injecting HTML elements embedding JS code or invoking built-in functions that enable JS execution.

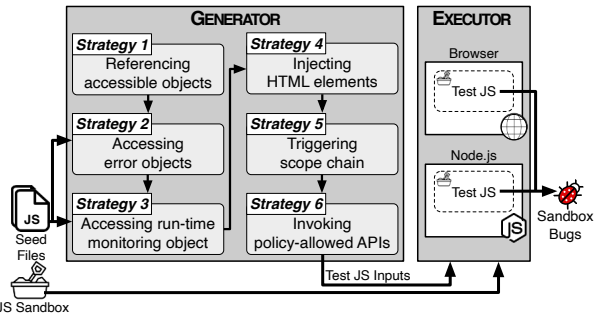

Figure 6: SandVenture architecture.

3.2. Limitations of Previous Work

Despite the importance and widespread adoption of JS sandbox frameworks, little research has been conducted on identifying their security flaws. Recent work [26] proposed SandDriller, the first JS sandbox testing framework tailored to test language-based sandboxes. It leverages the ECMAScript conformance test and V8 unit tests as a seed set. In particular, it interposes the oracle checks on all the references (e.g., return values) in the seed file to detect any foreign references. For example, if one of the testing JS codes is `var ret = func()`, SandDriller instruments the `ret` object to examine whether a foreign prototype chain can be made from it. By applying this method to six sandbox libraries, SandDriller identified 12 sandbox-escaping bugs.

Unfortunately, this approach has two key limitations. First, its effectiveness heavily depends on the built-in objects referenced in the original seed files. Since SandDriller only performs oracle checks on built-in objects in the given JS tests, it may miss vulnerabilities that involve less common or deeply nested objects. Second, it cannot detect vulnerabilities that occur through HTML injection, such as embedding JS code into DOM elements (Figure 5). This is because regression test cases, which SandDriller relies on, are limited to standalone JS snippets and do not cover HTML documents that embed JS code. Notably, such client-side attack vectors account for 26% of the reported bugs in our preliminary study.

Our approach. To address these limitations, we propose SandVenture, a novel fuzzing framework for discovering sandbox-escaping bugs. SandVenture is guided by concrete input generation objectives derived from our observations in a preliminary study (§3.1). In particular, SandVenture expands the testing scope in two key directions. First, it automatically explores diverse reference paths to built-in objects which increases the chance of discovering overlooked vulnerabilities. In addition, it implements mutational strategies to systematically generate a wide range of JS test inputs beyond those found in regression test suites. Second, it extends the testing to client-side sandbox frameworks by injecting HTML elements that embed JS code from within guest JS inputs, enabling the detection of HTML-based sandbox escapes.

4. Design

SandVenture takes in a sandbox under test and JS seed files. It then initiates a fuzzing campaign. During

the campaign, SandVenture generates a set of adversarial guest code (i.e., JS inputs) by following the input generation strategies that reflect our observations (§3.1). Each generated guest code attempts to bypass sandbox protection such that the sandbox under testing allows this JS input to access the host resources. Once the campaign is over, SandVenture reports a set of JS inputs that trigger sandbox-escaping vulnerabilities.

Figure 6 shows an overview of SandVenture. It consists of two components: the GENERATOR and EXECUTOR. At a high level, these components work together to conduct two steps: (1) the GENERATOR generates a set of JS inputs according to six generation strategies (§4.1), and (2) the EXECUTOR coordinates the testing of a client- or server-side sandbox to execute each JS input in the sandbox context of the browser or Node.js environment (§4.3). The EXECUTOR also checks that each testing JS input tampers with the host scope to confirm identified bugs in the sandbox under testing.

4.1. GENERATOR

The GENERATOR creates a set of JS inputs, each of which is a guest code for sandbox testing. Our goal here is to generate diverse forms of guest JS inputs that reflect our two observations (§3.1). Specifically, we define two objectives for sandbox testing, which reflect the observations: (1) obtaining references of diverse sandbox or browser built-in objects and (2) injecting JS code via policy-allowed bridges.

4.1.1. Objective 1: Referencing Diverse Built-in Objects. Our objective is to generate a set of JS inputs that reference diverse built-in objects. Specifically, we propose three generation strategies, each of which is designed to populate a different category of built-in objects. For each strategy, the GENERATOR creates a set of JS inputs by following the selected strategy. For each generated JS snippet, GENERATOR wraps the guest code with a validation snippet that corresponds to each strategy, as shown in Table 1. Note that those validation scripts are later used by the bug oracle (§4.3) to confirm the presence of sandbox-escaping bugs. We summarize three generation strategies below.

Strategy 1. Referencing accessible built-in objects. The first strategy is to reference normal built-in objects that are accessible from the guest scope. The GENERATOR enumerates a set of accessible built-in objects by traversing the properties from a given root object in a depth-first manner. By conducting this one-time enumeration process, the GENERATOR compiles a list of all accessible built-in objects, each of which is encoded with a reference path (e.g., `global.Object.name`) to access its corresponding object. After this one-time enumeration step, the GENERATOR inserts each generated reference path at the `[GENERATED_SCRIPT]` point in the validation script in the first row of Table 1 (pattern of Figure 3), thus generating a test input for each reference path.

In detail, the GENERATOR starts by executing an enumeration script that enumerates all accessible properties. The goal of this script is to (1) identify all reference paths stemming from a given object and (2) check the type and accessibility of each identified reference path. To achieve this, the script conducts a depth-first traversal from the `global` object [45], which is globally accessible from the guest scope. This object exists in every one of the 11 sandbox frameworks in our benchmarks (§5.1). To identify accessible properties belonging to this object, the enumeration script invokes the `getOwnPropertyNames` function [46] and checks the type of each identified property by using the `typeof` operator [47]. When the identified type is `object`, it records a reference path to this property and recursively conducts depth-first traversal from this identified property. We defined the maximum depth of traversal as three.

Strategies	Wrapping Methods
Strategy 1	`let referenced_obj = [GENERATED_SCRIPT]` `verify(referenced_obj);`

Strategy 2	`try {`
	`  [GENERATED_SCRIPT]`
	`} catch (referenced_obj) {`
	`  verify(referenced_obj);`
	`}`

Strategy 3	`Error.prepareStackTrace =(e,referenced_obj)=>{`
	`  verify(referenced_obj);`
	`}`
	`[GENERATED_SCRIPT]`

TABLE 1: Methods for wrapping scripts in Strategies 1, 2, and 3.

For example, the GENERATOR collects all accessible built-in objects from the `global` object as follows:

```
1  global  //root object
2   ├ global.Object           //getOwnPropertyNames(global)
3   │ ├ global.Object.length //getOwnPropertyNames(Object)
4   │ ├ global.Object.name
5   │   ...
6   ├ global.Proxy            //getOwnPropertyNames(global)
7   │ ├ global.Proxy.revocable //typeof(revocable)==func
8   │ ├ global.Proxy.revocable({}, {})
9   │ ├ global.Proxy.revocable({}, {}).revoke
10  ├ ...
```

Note that the enumeration script can encounter a `function` object while exploring all accessible properties. In this case, we need to access the properties of a returning object by invoking this function object with valid parameters. This enables us to cover built-in objects reachable via referencing any properties of such returning objects (e.g., the `revoke` property in Line 9). For this, the script obtains the length of the formal parameters via its `length` property and assigns one of the five predefined values [48] for each parameter: `'test'` (string), `123` (number), `true` (boolean), `{}` (object), and `()=>{}` (function). The script tries all combinations of these parameters to invoke the corresponding function object without raising errors, which attempts to exhaustively populate all reference paths containing function calls as shown in Line 9. If a function call with a particular combination of arguments does not produce an error (e.g., `TypeError` [49]), the script writes a reference path for invoking this function and continues the traversal from the returned object of this function call.

The GENERATOR generates approximately 400K JS inputs, each of which accesses a different reference path and wraps this reference path with the validation script, which vets whether its prototype chain contains a reference accessing the host scope (§4.3).

Strategy 2. Accessing error objects. The next strategy is to leverage built-in error objects [50] that a `catch`

```
1 var Realm = require('realms-shim');
2 let realm = Realm.makeRootRealm();
3 let guest_code = `
4   Error.prepareStackTrace = function (error, stack) {
5     stack.__proto__.__proto__.polluted = 'success'
6   };
7   x; ` // ReferenceError: x is not defined
8 realm.evaluate(guest_code);
9 console.log(polluted); // 'success' is printed
```

Figure 7: Sandbox-escaping vulnerability (CVE-2021-23543) in realms-shim reflecting Strategy 3.

block captures as an exception. For this strategy, we design the GENERATOR to create JS snippets that throw diverse types of error objects. We implement two approaches: (1) accessing host built-in objects from guest code and (2) leveraging regression test suites. For each generated JS snippet, by following these two approaches, SandVenture wraps this code with a `try-catch` expression. SandVenture then makes an error object caught by the `catch` block and validates this exception object, as shown in the second row of Table 1 (pattern of Figure 4).

Accessing host objects. For this, the GENERATOR compiles a list of built-in objects from the host scope by conducting the same procedure as in the first strategy. For each reference path in this list, the GENERATOR attempts to access this reference path from the guest scope and then collects reference paths that raise errors. The GENERATOR wraps each of these error-causing reference paths with a `try-catch` block, thus generating a set of JS inputs. For SandTrap, the GENERATOR identified 226,430 reference paths that raised exceptions in the guest scope, which we have leveraged to compile a set of JS inputs.

Leveraging regression test suites. We reuse a set of JS snippets from two types of public sources: (1) regression tests from the source repositories of four JS engines: ChakraCore, JavaScriptCore, V8, and SpiderMonkey, and (2) PoCs of JS CVEs [51]. We collected a total of 160K unique JS snippets, including 159 exploit files that trigger JS engine CVEs (§5.1). The GENERATOR wraps each of these code snippets with a `try-catch` block, thus validating the error objects caught by the `catch` block. The key idea here is to leverage regression tests that throw diverse types of error built-in objects, which SandVenture aims to exploit.

Strategy 3. Accessing run-time monitoring objects. The last strategy leverages run-time monitoring objects. We observed that many sandbox-escaping bugs abuse run-time monitoring objects (i.e., call stack objects) by customizing the `prepareStackTrace` method when an error occurs [52], [53], [54].

Figure 7 shows a PoC that successfully triggers a sandbox-escaping bug against one of the server-side sandbox frameworks, realms-shim, using the runtime-monitoring object. When an error occurs in guest code in Line 7 by referring to a non-existent variable `x`, a `stack` object is generated from the host scope when compiling the stack trace (Line 4), which makes the caller of `prepareStackTrace` the host scope. This immediately enables the JS code in the function body to access host resources, escaping sandbox protection.

For this, we generate JS inputs by prepending customized `prepareStackTrace` as shown in Table 1 to the JS inputs generated by following Strategy 2. That is, we reuse diverse error-triggering inputs but exploit the timing at which the call stack information is assembled because of triggered errors.

Steps	JS Snippets Being Combined

Step 1: Create element	`let element = document.createElement('script');`
	`let element = document.createElement('a');`
	`let element = document.createElement('img');`
	`let element = document.createElement('div');`
	`let element = document.createElement('iframe');`
	`...`

Step 2: Insert script	`element.innerText = '[VALIDATION_SCRIPT]';`
	`element.href = 'javascript:[VALIDATION_SCRIPT]';`
	`element.href = 'JaVaScript:[VALIDATION_SCRIPT]';`
	`element.src = 'javascript:[VALIDATION_SCRIPT]';`
	`element.onload = '[VALIDATION_SCRIPT]';`
	`element.style = 'background-image: url(javascript:[VALIDATION_SCRIPT]';`
	`...`

Step 3: Append element	`document.body.appendChild(element);`
	`document.body.insertBefore(element, null);`
	`...`

TABLE 2: Generation rules used in Strategies 4 and 5.

According to the three strategies, the GENERATOR generates a total of 833,411 JS inputs, which become the seed inputs for further mutation to maximize the chance of identifying sandbox-escaping bugs. We describe the mutation process of SandVenture in §4.2.

Validation script. The goal of the validation script is to add the `polluted` property to every object reference on the prototype chains that stem from a given object. When this `polluted` property appears in any built-in objects in the host scope, the generated JS input triggers a bug, breaking the isolation of guest code [26], [38].

4.1.2. Objective 2: Injecting JS Code via Host Object.

We design three generation strategies under this objective. These strategies include injecting HTML elements that embed JS code into the host scope, exploring scope chain resolution to access host variables, and invoking host-exposed built-in functions that enable JS execution.

Strategy 4. Injecting HTML elements. We observed that several sandboxes allow access to DOM elements with limited privileges according to sandbox policies [9], [11], [12], [44], [55]. For example, in TreeHouse and Caja, HTML injection for script executable vectors such as the `script` tag or the `onload` attribute from guest code is blocked through the default policy [56], [57].

In order to bypass a given policy, so that guest code can inject HTML elements that embed JS snippets into the host context (e.g., Figure 5), the GENERATOR generates JS inputs that append HTML elements in various forms. In particular, the GENERATOR generates all known HTML forms of executing JS snippets by referencing the ECMAScript specification [58] and known XSS attack payloads [59].

Table 2 shows the test code generation method used in Strategy 4. For each identified HTML element (Step 1), the GENERATOR inserts a validation JS snippet into the attributes or inner text in various forms (Step 2). It then attempts to inject this element into the host scope via policy-allowed bridges (Step 3).

Strategy 5. Triggering scope chain. The JS scope chain is a runtime mechanism that locates a variable's declaration [60]. If a variable is not declared in the current scope, the engine searches outward through enclosing scope, such as outer blocks, until it finds the correspond-

Type	Allowed API	Test Case
Dynamic code conversion	eval	`eval('[VALIDATION_SCRIPT]');`
	Function	`Function('[VALIDATION_SCRIPT]')();`
	Timeout	`setTimeout('[VALIDATION_SCRIPT]');`
	Interval	`setInterval('[VALIDATION_SCRIPT]');`
Dynamic page redirection	open	`open('javascript:[VALIDATION_SCRIPT]');`
	replace	`location.replace('javascript:[VALIDATION_SCRIPT]');`
Module import	import	`import('data:application/javascript,[VALIDATION_SCRIPT]');`

TABLE 3: Allowed APIs that can dynamically execute JS code.

ing declaration. In a sandboxed environment, this chain must be properly filtered to enforce isolation. Otherwise, if the guest code references an undeclared variable, the engine may resolve it to one in the host scope, potentially breaking the sandbox boundary.

In Strategy 5, SandVenture examines whether scope chain filtering is correctly enforced across various objects. Specifically, it inserts the Step 1 and Step 3 snippets from Table 2 into the host code to create an HTML element. Then, it inserts the Step 2 snippet into the guest code. Since the `element` variable is declared only in the host scope, any reference to it by guest code will trigger a scope chain lookup. If the sandbox filters the scope chain correctly, this access results in a reference error. Otherwise, the guest code may resolve the variable and gain unauthorized access to the host scope.

Note that Strategy 5 differs from Strategy 4 in terms of the targeted escaping pattern. Strategy 4 targets weak DOM-element filtering (e.g., Figure 5 and Figure 11) by leveraging the DOM APIs intentionally exposed within the guest context, whereas Strategy 5 targets weak scope-chain filtering (e.g., Figure 12) by attempting to access the host's DOM APIs even though they are not exposed within the guest context.

Strategy 6. Invoking policy-allowed APIs. Client-side sandbox frameworks support specifying a policy that lists host built-in functions accessible from guest code in an as-is manner, a mechanism commonly referred to as endowment. For example, in the jailed sandbox, a user may allow the `setTimeout` API using `new jailed.Plugin('[JS_PATH]', {setTimeout:setTimeout})` to enable timer functionality. However, an attacker can exploit this allowance to execute arbitrary JS code in the host context, deviating from the original intent.

To evaluate the security implications of such configurations, we define a set of permissive policies for each sandbox framework to allow JS-executable built-in functions. Table 3 shows the list of allowed APIs known to support dynamic JS execution [58] and their test cases used by SandVenture. For each API, the GENERATOR generates a JS input that invokes each specified built-in function with a JS payload (Third column in Table 3).

Although certain sandboxes proactively block code execution attempts via allowed APIs, it is ultimately the host developer's responsibility to configure policies securely. Since such behavior results from insecure configuration rather than flaws in the sandbox itself, we count these cases as sandbox misuse, not vulnerabilities. Nevertheless, our goal is to systematically generate JS inputs that break sandbox guarantees. We believe that the identified misuse cases can help document potential risks in sandbox manuals and encourage safer configurations.

Algorithm 1: Mutation-based Fuzzing Algorithm.

```
1  function Fuzzing(seed_corpus, sandbox)
2      seed_ast_corpus ← Codes2AST(seed_corpus)
3      while True do
4          seed_ast ← PickRandSeed(seed_ast_corpus)
5          mutation_op ← PickRandOperation()
6          switch mutation_op do
7              case "replace_subtree" do
8                  location, node_type ← PickRandNode(seed_ast)
9                  subtree ← PickRandSubTree(seed_ast_corpus,
                     node_type)
10                 mutated_ast ← ReplaceSubtree(seed_ast, location,
                     subtree)
11             case "insert_statements" do
12                 location ← PickRandStat(seed_ast)
13                 stat_blk ← PickRandStatBlk(seed_ast_corpus)
14                 mutated_ast ← InsertStatBlk(seed_ast, location,
                     stat_blk)
15         mutated_code ← AST2Code(mutated_ast)
16         mutated_code ← WrapCode(mutated_code)
17         EXECUTOR(mutated_code, sandbox)
```

Validation script. For bug validation in a client-side sandbox, the validation script attempts to inject an arbitrary key-value pair `"flag=flag"` into the document.cookie. When a given JS input escapes the guest scope, the content of the cookie in the host document environment differs from the initial value.

4.2. Mutation-based Input Generation

SandVenture conducts mutations on JS seed files to generate diverse JS inputs, diversifying the ways of accessing built-in objects (Objective 1).

Seed corpus. As a seed corpus, SandVenture leverages 160K JS files consisting of the regression tests and JS engine CVEs that we use for implementing Strategy 2. In addition, we also included 115 files consisting of the JS sandbox regression tests and the PoCs of CVEs that trigger known sandbox-escaping vulnerabilities. In total, we prepared 161,038 unique JS files as the seed corpus. We demonstrate the efficacy of each seed type in finding bugs in §5.3.

Algorithm 1 describes the overall fuzzing procedure. The `Fuzzing` function takes in a set of seed files (*seed_corpus*) and sandboxes under testing (*sandbox*). It then begins by transforming each seed file into an abstract syntax tree (AST) in Line 2. The reason we transform JS code into AST is to efficiently mutate code while maintaining its syntax structures, similar to common JS fuzzers [61], [62], [63]. After that, SandVenture enters a fuzzing loop in Lines 3–17. It starts by randomly selecting a seed AST from the given set (Line 4) and a mutation operation (Line 5). Lines 6–14 then mutate the seed AST according to the selected mutation operation and output the mutated AST (*mutated_ast*). These steps enable the composition of a new JS input in Line 15. The generated input is then wrapped in the validation script to reflect strategies 1–3 (Line 16). Finally, the wrapped code is passed to the EXECUTOR so that it can be tested in each sandbox (Line 17).

Mutation operation. SandVenture adopts two mutation operations: (1) replacing an AST subtree and (2) inserting a block of statements.

To replace an AST subtree, SandVenture randomly selects an AST node from the selected seed AST (Line 8). The AST subtree rooted at the selected node is then replaced with a new one randomly selected from the subtrees in the seed corpus (Lines 9–10). In order to maintain the semantics of execution here, only subtrees whose type of root node (e.g., call expression) is the same as the type of root node being replaced are selected.

To insert a statement block, SandVenture randomly selects one of the top-level statements (ended with a semicolon) from the seed AST for insertion (Line 12). Next, Line 13 randomly selects a block of top-level statements from the seed corpus. SandVenture then inserts the selected block at the end of the selected statement (Line 14).

These two operations are widely used in JS fuzzers to discover memory-level bugs [61], [62], [64], [65]. Although our goal is to find sandbox-escaping vulnerabilities, we leverage these operations to diversify the patterns for accessing JS objects. This design choice is inspired by our observation that PoC exploits for sandbox-escaping bugs often share common code fragments when varying their object-access patterns [66], [67], [68].

Code wrapping. In Line 16, SandVenture wraps the mutated input using the validation scripts used in Strategies 2 and 3 at the same time, producing two input files for this mutated input. This approach expects the mutated input to trigger diverse errors that can be caught in the validation scripts and exploited to escape the sandbox. In addition, to expand Strategy 1, we collect all objects referenced by each top-level statement and then wrap each object with the validation script, thus producing testing files.

4.3. EXECUTOR

For each guest JS input generated by the GENERATOR, the EXECUTOR executes the input in an isolated environment for each sandbox. In particular, the server-side sandboxes are tested in the Node.js environment, and the client-side sandboxes are tested in the Chromium browser environment. Also, for the set of JS execution APIs that require endowment by specifying a sandbox policy required to achieve the second objective (§4.1.2), we specify the corresponding APIs using the endowment function provided by each sandbox.

Bug oracle. We implement different bug oracles for server-side sandboxes and client-side sandboxes to verify whether a given JS input contributes to escaping the sandbox under testing. After running the test JS code in the server-side sandbox environment, the EXECUTOR verifies the existence of the `polluted` property in any built-in objects in the host scope. For the bug oracle for client-side sandboxes, the EXECUTOR checks whether the cookie value of the host document is `"flag=flag"` to verify the existence of a bug.

Test case minimization. Considering that we observed 48,721 test cases that successfully passed the bug oracle, the manual analysis of each case to identify its root causes would be intractable. Accordingly, we leveraged delta debugging [69] to deduplicate the number of bugs. In particular, given a test case that successfully passed the bug oracle, the EXECUTOR shrinks the corresponding test case by iteratively deleting statements, functions, and sub-expressions, looking for a small subprogram that still triggers sandbox-escaping bugs.

Sandbox (Version)	Popularity	Type Client	Type Server	# of Bugs Client	# of Bugs Server	# of Bugs Total
vm2 (3.9.14)	95K ⭳ / 3.9K ★	-	✓	-	3	3
safe-eval (0.4.1)	32K ⭳ / 264 ★	-	✓	-	18	18
jailed† (0.3.1)	94 ⭳ / 1k ★	✓	✓	0	19	19
SandTrap (1.0.3) [8]	-	-	✓	-	6	6
JS-Interpreter (4.0.0)	3K ⭳ / 2.1K ★	✓	✓	0	0	0
TreeHouse† [9]	-	✓	-	1	-	1
isolated-vm (4.5.0)	299k ⭳ / 2.4K ★	-	✓	-	0	0
ADsafe† (43cd3dd)	238 ★	✓	-	3	-	3
SES (0.12.2)	90K ⭳ / 876 ★	✓	✓	3	0	3
near-membrane (0.12.13)	19 ⭳ / 119 ★	✓	✓	0	16	16
sandbox (0.8.6)	712 ⭳ / 851 ★	-	✓	-	16	16
Total				7	78	85

⭳: # of weekly downloads on npm
★: # of stars on GitHub
† # of no update for 10+ years

TABLE 4: Number of sandbox-escaping bugs found with SandVenture.

5. Evaluation

We evaluate the capability of SandVenture in discovering sandbox-escaping vulnerabilities (§5.2) and analyze the efficacy of the implemented test generation strategies and mutation operations in identifying bugs (§5.3). We also compare SandVenture with SandDriller (§5.4), and present case studies of the discovered bugs (§5.5).

5.1. Experimental Setup

We conducted experiments on one machine running 64-bit Ubuntu 22.04.5 with 160 CPUs and 512 GB of main memory. To test client-side sandboxes, we developed a test automation script using Playwright 1.51.0 [70], which loads each sandbox default code that confines a generated JS input and runs it with Chromium version 134. We set an execution timeout of two seconds for each sandbox execution.

JS sandboxes. We ran a series of experiments on the 11 JS sandboxes listed in the first column of Table 4. We selected our benchmark sandboxes from the three sources: (1) highly rated npm libraries; (2) highly rated GitHub repositories; and (3) the evaluation set used by SandDriller [26], which reported no errors during installations. We also selected three open-source sandbox prototypes [8], [9], [55] that were published in prestigious academic conferences.

Seed corpus. As SandVenture mutates existing test suites and PoCs, the quality of the seed corpus affects fuzzing performance. To build the seed corpus of SandVenture, we collected 160,923 regression tests from the source repositories of four JS engines: ChakraCore, JavaScriptCore, V8, and SpiderMonkey, and PoCs of JS CVEs [51]. In addition, we prepared 115 files consisting of the JS sandbox regression tests [71] and the PoCs of 18 CVEs that trigger known sandbox-escaping vulnerabilities. In total, we prepared 161,038 unique JS files used in the following experiments.

5.2. Real-World Bugs Found

We evaluated the capability of SandVenture in finding real-world bugs. Table 4 describes the experimental

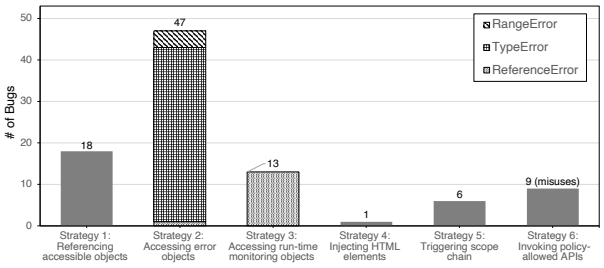

Figure 8: Number of sandbox-escaping bugs found per input generation strategy.

results. SandVenture found 85 distinct sandbox-escaping bugs, including 78 server-side bugs and seven client-side sandbox bugs from 10 libraries. Table 8 in Appendix describes the details of all bugs and misuse cases found by SandVenture.

All the discovered vulnerabilities have critical security impacts on their corresponding sandboxes. The adversary exploiting these vulnerabilities is able to execute arbitrary JS code in the host scope, which can lead to remote code execution or host DOM manipulation.

Bug counting method. Given a set of JS inputs that break a target sandbox, we conducted delta debugging [72] to deduplicate JS inputs. Through this approach, we minimized the 48,721 test cases that successfully passed the bug oracle, resulting in 1,145 test cases. However, a portion of the 1,145 cases still shared the same root causes while differing in minor syntactic details (e.g., constant values or block locations). To count the unique number of bugs, we clustered these minimized JS inputs based on the objects or APIs they accessed for sandbox escaping, resulting in 85 vulnerabilities and nine misuse cases.

Bug disclosure. Among 85 vulnerabilities, four vulnerabilities from three sandboxes had already been patched or reported at the time SandVenture discovered them. We reported the remaining 81 vulnerabilities to their corresponding vendors and received four CVEs from three vendors. Among the reported cases, 11 vulnerabilities from four vendors have been patched. One vendor acknowledged the seven vulnerabilities and stated that they plan to deprecate the sandbox. For the remaining 63 vulnerabilities, we are waiting for their responses.

Recall from §4 that we defined six strategies, each of which is designed to exhibit a different generation pattern. We counted the number of bugs triggered by JS inputs that each generation strategy creates. Figure 8 presents the number of vulnerabilities triggered by each generation strategy, which determines a bug type.

Referencing accessible objects. We observed that 18 out of 85 bugs (19%) were due to simple references to accessible built-in objects. In particular, we noticed that several sandboxes, including safe-eval, jailed, sandbox, and near-membrane, do not properly tame global built-in objects, including `this`, `globalThis`, `__proto__`, and `constructor`. Considering the prevalence of these vulnerable sandboxes, the identification of these vulnerabilities certainly contributes to improving the security of those sandboxes, impacting a large number of users.

SandTrap and vm2 prevent the aforementioned vulnerability by replacing referenced objects with guest objects when the referenced objects belong to the host scope. However, through our mutation fuzzing process, SandVenture found that SandTrap also exposed the `this` object when this object is polluted in the execution context of `eval`, such as `eval("arg=>verify(arg)")(this)`. The discussions with vendors revealed that SandTrap exhibits shortcomings in isolating the `eval` execution context introduced in version 1.0.3.

Accessing error objects. We observed that the majority of bugs (47 out of 85 bugs–50%) were triggered by error objects populated by Strategy 2. In particular, among the types of errors, we observed that most vulnerabilities (41 out of 47 bugs) were triggered by JS inputs throwing `TypeError` objects. We confirmed that several built-in function calls such as `propertyIsEnumerable`, `valueOf`, and `toLocaleString`, threw `TypeError` exceptions, leading to vulnerabilities. Note that the function objects accessible via these properties are directly callable, and the returned objects of these calls have prototype chains having references to the host scope. Many sandboxes focused on protecting properties that are derived from the global object, but often missed protection for returning objects of these functions.

Accessing run-time monitoring objects. We confirmed that 13 bugs were related to run-time monitoring objects. All the bugs were triggered by `ReferenceError`, but after further analysis, we observed that the `stack` object becomes the host object depending on where the error is thrown, regardless of the type of error. For example, we observed in vm2, safe-eval, sandbox, and near-membrane that the `stack` object in the `prepareStackTrace` function becomes a host object if any errors are thrown into the `async` function. Sandbox maintainers need to tightly control this sandbox-escaping path by properly isolating the `prepareStackTrace` method.

Injecting HTML elements. SandVenture succeeded in bypassing the TreeHouse sandboxing policy by applying Strategy 4. In particular, we observed that the logic of determining whether to append an HTML element by detecting the `javascript:` string can be bypassed through a JS URL containing uppercase letters (e.g., `JaVaScript:`). Through this vulnerability, the guest code running on the web worker can successfully inject HTML embedding JS code that can be executed in the host context. SandVenture discovered this bug while injecting various HTML forms of executing the JS code into the host DOM environment.

Triggering scope chain. SandVenture found six vulnerabilities related to incomplete scope chain filtering in SES and ADsafe. In particular, both frameworks failed to properly filter scope chains for variables declared in lexical environments [73] (i.e., variables declared using `let`, `const`, or `class`) introduced in ECMAScript 6. These flaws allow an attacker to access the host environment by referencing variables that are declared in the host but not in the guest. The SES team patched the vulnerabilities and rewarded us with a bug bounty of 5,000 USD.

Invoking policy-allowed APIs. For reports caused by Strategy 6, we deduplicated the number of bugs by grouping test cases by function type in Table 3. As a result of deduplication, we found nine potential misuse cases in jailed, JS-Interpreter, and SES by abusing the allowed set of APIs.

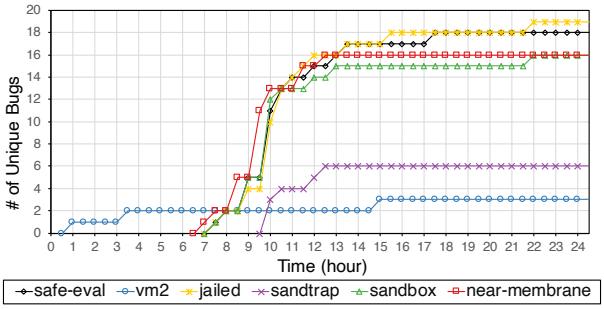

Figure 9: Cumulative numbers of sandbox-escaping bugs in server-side sandboxes over time.

We reported these misuse cases to their respective vendors, and the maintainers of JS-Interpreter and SES acknowledged that allowing host APIs makes the sandbox vulnerable. Furthermore, we encouraged each vendor to offer comprehensive documentation to help sandbox users avoid potential misuse of the exposed APIs.

Performance. For frameworks that support only server-side sandboxing (e.g., vm2), we ran SandVenture with a 24-hour (3,840 CPU hours) timeout per sandbox, applying the testing strategies for Objective 1 together with the mutations. For frameworks that support only client-side sandboxing (e.g., TreeHouse), testing was completed within 10 minutes (26 CPU hours) per sandbox, as SandVenture only needs to feed the predefined combinations of test cases proposed to achieve Objective 2. For frameworks that provide both client-side and server-side sandboxing (e.g., jailed), we applied all testing strategies for Objectives 1 and 2, along with the mutations, under a 24-hour timeout.

Figure 9 presents the cumulative number of unique bugs in server-side sandboxes over the testing time. As the figure shows, to find 50% of unique bugs, SandVenture required less than 10 hours. For 90% of the bugs, it required approximately 13 hours. We observed that nine bugs were found after 12 hours when conducting mutations.

5.3. Ablation Studies

SandVenture takes in two kinds of seed files and leverages two mutation operations (§4.1.1). We evaluated the effectiveness of each technical component by running SandVenture without the corresponding component. In particular, we ran SandVenture without each component listed in Table 6 and counted bugs discovered within 24 hours. This enables measuring the efficacy of each component with respect to identifying sandbox bugs.

Table 6 shows the number of bugs found in nine server-side sandboxes using five different approaches, i.e., SandVenture, *w/o* seed #1, *w/o* seed #2, *w/o* mutation operation #1, and *w/o* mutation operation #2. Each approach found a subset of the 78 bugs identified in §5.2. In particular, we confirmed that finding more bugs is determined by the presence or absence of the mutation operation type rather than the presence or absence of the seed type. This result highlights that the quality of mutation operation affects the performance of SandVenture [74], [75]. We concluded that combining all seed components with mutation techniques yields the most effective results.

```
const VM = require("vm2");
let vmInstance = new VM();
const guest_code = `
  Error.prepareStackTrace = (e, stack) => {
      stack.constructor.constructor('return process')()
                .mainModule.require('child_process')
                .execSync('touch flag');
  };  //Wrapping code according to Strategy 3
  async function func1 () {
    eval("x");//Inserted statement: throw ReferenceError
  }
  func1(); `
vmInstance.run(guest_code);
```

Figure 10: A test case generated by SandVenture triggering CVE-2023-29017 of vm2@3.9.14.

5.4. Comparison against SandDriller

We compared SandVenture with the state-of-the-art sandbox testing tool, SandDriller [76]. We counted the unique number of vulnerabilities each tool found. In this evaluation, to ensure a fair comparison, we excluded ADsafe, near-membrane, and safe-eval from the benchmarks, as they were known to be broken [26]. We also did not count cases corresponding to Strategy 6 because they depend on usage and cannot be considered pure vulnerabilities. Consequently, we ran both tools on eight sandboxes, which contained a total of 48 bugs (refer to Table 5), for 24 hours using the same seed corpus (i.e., V8 and Test262).

Table 5 shows the detailed results of the comparison between SandVenture and SandDriller. The third column presents the details of the 48 bugs, with each bug separated by " / ". The fifth and sixth columns indicate whether each tool detects the corresponding bug(s).

We observed that SandVenture discovered 48 vulnerabilities, whereas SandDriller found 33 vulnerabilities. In particular, our mutation-based fuzzing methods reflecting Strategies 1 and 2 enabled SandVenture to uncover five more vulnerabilities. Furthermore, SandVenture identified six more bugs by leveraging Strategy 3. For example, a bug in SandTrap (1.0.3), the pattern of assigning a value to `Object.property.set`, does not exist in the seed corpus in its original form and therefore cannot be covered by SandDriller. This is the pattern obtained by changing an AST subtree through our mutational fuzzing. Furthermore, SandVenture discovered four more vulnerabilities by applying Strategies 4 and 5, which target HTML injection vectors and scope chain.

In summary, SandDriller missed 15 vulnerabilities due to (1) a limited search space related to Objective 1 (11 bugs) and (2) the lack of consideration for Objective 2 (four bugs). In contrast, SandVenture generates inputs guided by distinct and well-defined objectives, allowing it to explore a wider space of potential vulnerabilities. Importantly, SandVenture also identified all bugs found by SandDriller, demonstrating that our approach extends the coverage of existing techniques.

5.5. Case Studies for the Bugs Found

We investigate the findings of SandVenture during the experiments (§5.2) and how our approach contributed to uncovering bugs. We simplified the test case for ease of explanation.

vm2. Figure 10 shows a test case generated by SandVenture, which triggers CVE-2023-29017 on vm2 [34].

Strategy	Sandbox	†Vulnerability Details	# of Bugs	SandVenture	SandDriller
Strategy 1	jailed	- Access to `import` / `this` / `Proxy(this, this)` / `constructor`	4	✓	✓
	SandTrap	- Access to `this` from the `eval` context	1	✓	✗ (1)
	sandbox	- Access to `import` / `this` / `Proxy(this, this)` / `constructor`	4	✓	✓
Strategy 2	jailed	- Recursion with `Error` stack calling (`RangeError`)	1	✓	✓
		- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2	✓	✓
		`__lookupGetter__()` / `__lookupSetter__()` / `toString()`	3	✓	✓
		`hasOwnProperty()` / `isPrototypeOf()` / `propertyIsEnumerable()` / `valueOf()`	4	✓	✓
		`setTimeout()` / `clearTimeout()` / `setInterval()` / `clearInterval()` (`TypeError`)	4	✓	✓
	SandTrap	- Recursion with `Error` stack calling (`RangeError`)	1	✓	✓
		- Error thrown by assigning a value to `eval.toString` (`ReferenceError`)	1	✓	✗ (1)
		`Object.property.set` / `Object.property.get` (`TypeError`)	2	✓	✗ (2)
		- Error thrown by deleting `Array.prototype[Symbol.iterator]` (`TypeError`)	1	✓	✗ (1)
	sandbox	- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2	✓	✓
		`__lookupGetter__()` / `__lookupSetter__()` / `toString()`/ `valueOf()`	4	✓	✓
		`hasOwnProperty()` / `isPrototypeOf()` / `propertyIsEnumerable()` (`TypeError`)	3	✓	✓
Strategy 3	jailed	- Error thrown in the `promise` context	1	✓	✗ (1)
	vm2	- Error thrown in the `async` / `promise` / `WebAssembly` context	3	✓	✗ (3)
	sandbox	- Error thrown in the `global` context	1	✓	✓
		- Error thrown in the `async` / `promise` context	2	✓	✗ (2)
Strategy 4	TreeHouse	- JS URL containing uppercase letters (i.e., `JaVaScript:`)	1	✓	✗ (1)
Strategy 5	SES	- Abuse by scope chain `let` / `const` / `class`	3	✓	✗ (3)
		Total	48	**48**	33

† Each bug is separated by " / " ✓: The tool found all bugs in that row ✗ (n): The tool missed all bugs in that row (n: # of missed bugs)

TABLE 5: Comparison between SandVenture and SandDriller with benchmarks.

w/o	# of Reported Bugs	# of Unique Bugs
SandVenture (all inclusive)	48,721	**78**
Seed #1: JS regression tests	13,798	62
Seed #2: JS sandbox bugs	30,787	68
OP #1: Replacing an AST subtree	38,995	61
OP #2: Inserting a block of statements	40,671	60

TABLE 6: Number of bugs found in server-side sandboxes using five different approaches.

We observed that the `stack` object in Line 4 becomes a host object if an error (`ReferenceError`) is thrown in `async()` [77] in Line 10. The maintainers of vm2 patched this vulnerability by wrapping the `prepareStackTrace` function to convert the `stack` object into a sandbox object when it is a host object.

Note that generating such a code snippet is not trivial as two conditions must be satisfied: (1) throwing an error in `async()` and (2) accessing the run-time monitoring object `stack` in `prepareStackTrace()` (Strategy 3). We observed that SandDriller cannot trigger this bug because no code that satisfies all of these conditions exists in the seed corpus.

SandVenture successfully generated the test case by mutating the seed JS file (§4.2). In particular, SandVenture satisfied the first condition by inserting a statement block that calls `eval()` in Line 10 inside the `async` function. In addition, SandVenture achieved the second condition by prepending customized `prepareStackTrace` (Strategy 3 in Table 1) in Line 4. This case demonstrates that SandVenture is capable of generating comprehensive inputs that reflect Strategy 3.

TreeHouse. TreeHouse [9] has a sandbox-escaping vulnerability that stems from the insecure sandboxing policy. Figures 11b and 11c show TreeHouse's default sandboxing policy for tags and attributes that can be injected into the host context, respectively. In TreeHouse, HTML injection for script executable vectors such as `script` tag (Line 2 in Figure 11b) or `onload` attribute (Line 3 in Figure 11c) from guest code is blocked by the default policy.

However, SandVenture successfully bypassed the

```
1 let element = document.createElement("a");
2 element.href = "JaVaScript:document.cookie=`flag=flag`";
3 document.body.appendChild(element);
```

(a) Guest code.

```
1 tag_policy: {
2   script: false,
3   iframe: false,
4   a: true,
5   object: false,
6   ...
7   }
```

(b) Tag policy.

```
1 attribute_policy: {
2   height: true,
3   onload: false,
4   ...
5   href: match(
6     new RegExp('^(?:javascript):.+'))
7 }
```

(c) Attribute policy.

Figure 11: A test case generated by SandVenture triggering a sandbox-escaping bug in TreeHouse.

```
1 let element = document.createElement("a")
2 document.body.appendChild(element)
3 const sandbox = new Compartment({});
4 const guest_code = `element.href="
5               javascript:document.cookie=`flag=flag`"`;
6 sandbox.evaluate(guest_code);
```

Figure 12: A simplified test case generated by SandVenture triggering CVE-2025-32792 of SES@1.12.0

given policy by leveraging the `href` attribute (Line 5 in Figure 11c) for the allowed `a` tag (Line 4 in Figure 11c). Figure 11a presents a guest code triggering a sandbox-escaping bug in TreeHouse. In particular, SandVenture creates an HTML element for the `a` tag in Line 1 and adjusts the `href` attribute with a JS URL containing uppercase letters (i.e., `JaVaScript:`) in Line 2, allowing the PoC to bypass JS URL inspection (Line 6 in Figure 11c). Through this exploit, the guest code running on the web worker can successfully inject HTML embedding JS code that can be executed in the host context. This study shows that it is effective to generate HTML injection code that embeds JS in various forms reflecting Strategy 4 (Table 2).

SES. Figure 12 illustrates a PoC exploit that demonstrates a scope chain traversal. In particular, the `element` variable referenced in Lines 4-5 is not declared within the guest scope. Consequently, the browser JS engine follows the scope chain and resolves it to the variable declared with `let` in the host scope (Line 1). SES fails to properly filter access to variables in the host's lex-

ical environment [73], thereby allowing guest code to modify host-level resources exposed through the global scope. To make matters worse, SES has weak filtering mechanisms for browser-specific attack vectors, such as the `javascript:` scheme, which makes it possible to execute malicious JS code in the host scope.

6. Limitations and Discussion

Input generation strategies. We presented six generation strategies that implement our two objectives that capture the characteristics of common sandbox-escaping bugs. We acknowledge that there may exist other patterns in JS inputs triggering sandbox bugs. For example, exploiting HTML parsing differentials [78] could reveal previously unknown sandbox-escape bugs in client-side sandboxes. However, we emphasize that the presented two objectives are general enough because these objectives are based on our comprehensive analysis of covering 67 known bugs. At the same time, SandVenture supports extensions of the current system, allowing users to add seed files or mutation operations to further exercise the capability of generating diverse inputs.

Adding new strategies. We acknowledge that covering new types of sandbox-breaking bugs necessitates manual engineering effort. It involves (1) clustering bugs, (2) identifying a common root cause within each cluster, (3) formulating a test generation objective for each identified cause, and (4) devising test generation strategies to achieve each objective. Automating these procedures requires further research. We note that the vast search space for finding bugs often causes JS fuzzers to generate test inputs with specific patterns, based on the authors' observations [62], [79], [80], [81].

Mutation-based input generation. We acknowledge that SandVenture exhibits redundancy in test cases that stem from the random mutation strategy, from which most JS fuzzers have suffered [82]. We believe that by adopting feedback-driven approaches, it is possible to address this issue [83]. Exploring the incorporation of these approaches for fuzzing JS sandbox libraries remains an open research question that we need to explore. SandVenture takes both generational and mutational fuzzing approaches. Therefore, SandVenture is able to identify 24 sandbox bugs without a given seed corpus (§5.2), demonstrating its capability of generating new seed inputs.

Sandbox deprecation. Recently, several language-based sandbox frameworks have been deprecated due to structural limitations [84], [85]. Most of these sandboxes implement isolation by interposing membranes on all exchanged objects. We argue that this approach is fundamentally flawed. The dynamic nature of JS, with newly introduced features and built-in objects, inevitably introduces loopholes in sandbox implementations. We believe SandVenture, with an updated regression test suite, can help identify edge-case objects, expose inherent weaknesses, and maintain sandbox security.

Possible mitigations. Based on the bugs found, we believe there is a need to develop more robust defense mechanisms. One possible mitigation is to add a dynamic monitoring system that observes any access to global objects and returns bogus objects in the guest scope. Another solution is to fundamentally prevent prototype pollution through runtime-based isolation and to block or monitor APIs that dynamically execute JS code.

We also observed that many sandbox-escaping vulnerabilities (47 bugs) were triggered by accessing error objects. This trend stems from the developer's mistake of not properly taming the error objects generated by various patterns of exceptions. Library developers should comprehensively tame various error objects in order to completely confine the guest code.

Policy for allowed APIs. We identified nine misuse cases. These cases do not constitute inherent vulnerabilities in the sandbox itself. However, existing sandboxes often allow host developers to expose host-side capabilities to guest code. To achieve stronger isolation, we recommend that sandbox maintainers either proactively block JS dynamic-execution APIs or ensure that dynamically executed code properly inherits the intended isolation scope. At a minimum, sandbox maintainers should clearly document the risks associated with these misuse cases.

7. Related Work

JS isolation. Prior studies have largely focused on studying the isolation of JS code [8], [9], [10], [11], [12], [30], [86], [87], [88]. SandTrap [8] uses `vm` and `proxy` modules to enforce access control policies on guest code. ScriptProtect [10] instruments problematic APIs, e.g., `document.write`, to ensure that guest code is unable to add unsafe markup into the host context. Several works have focused on isolating guest code along with the virtualized DOM elements [9], [11], [89]. TreeHouse [9] isolates guest code in a web worker and offers virtualized browser APIs to guest code. AdSentry [11] runs third-party JS code on the shadow JS engine sandboxed by Native Client [90]. AdJail [33] and SMash [91] leverage iframe elements to confine third-party code.

Another recent line of research focuses on analyzing and escaping isolation [26], [55], [92]. Politz *et al.* [55] proposed a static analysis for verifying the safety of language-based JS sandbox libraries [93]. Recently, Al-Hamdan and Staicu proposed SandDriller, a JS fuzzer designed to test language-based sandbox libraries [26]. They leveraged existing regression JS tests and interposed the oracle checks on all the references to identify foreign references.

Unlike these approaches, SandVenture is based on concrete objectives in input generation and leverages both generational and mutational approaches. This enables SandVenture to generate a more comprehensive set of JS inputs contributing to the discovery of 15 unique bugs that previous works were unable to find.

Adversarial JS generation. There has been a surge in research in generating adversarial JS inputs to test client/server-side web applications, including browser engines [79], [94] and Node.js applications [38], [95]. Kim - *et al.* [79] proposed FuzzOrigin, a browser fuzzer designed for finding universal cross-site scripting (UXSS) vulnerabilities. Shcherbakov *et al.* [38] proposed the static analysis framework for finding prototype pollution gadgets that cause remote code execution in Node.js applications.

Recently, there have been several studies on JS engine fuzzing [61], [62], [63], [64], [65], [82], [96]. CodeAlchemist [82] proposed semantics-aware assembly and

Fuzzilli [64] designed their own intermediate representation to generate syntactically and semantically valid JS code. DIE [62] leverages aspect-preserving mutation, which is the method that preserves interesting properties of input seed for generating test cases. We referred to the mutation operations they used, but we used them to generate JS input that meets the objective for testing against JS sandboxes, not browser JS engines.

8. Conclusion

We present SandVenture, a JS sandbox testing tool designed to identify sandbox-escaping bugs. We conducted the qualitative study of 67 known bugs and derived two observations from their commonalities. Based on these observations, we propose two objectives in JS input generation: (1) obtaining diverse built-in object references and (2) abusing host-guest bridges to inject JS code. We also propose six generation strategies that reflect these objectives, thus generating diverse JS inputs that are highly likely to trigger sandbox-escaping bugs. SandVenture found 85 sandbox-escaping bugs from 11 sandbox frameworks, demonstrating its efficacy in breaking security guarantees of real-world JS sandboxes.

Acknowledgments

We would like to thank the anonymous reviewers for their concrete feedback. This work was supported by Innovative Human Resource Development for Local Intellectualization program through the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2026-RS-2022-00156361) and National Research Foundation of Korea (NRF) (grant number: RS-2025-00561150).

References

[1] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote JavaScript inclusions," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012, pp. 736–747.

[2] T. Hou, S. Bi, M. Wei, T. Wang, Z. Lu, and Y. Liu, "When third-party JavaScript meets cache: Explosively amplifying security risks on the internet," in *Proceedings of the IEEE Conference on Communications and Network Security*, 2022, pp. 290–298.

[3] "Heroku," https://www.heroku.com/.

[4] "Fly.io," https://fly.io/.

[5] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[6] Y. Zhou and D. Evans, "Understanding and monitoring embedded web scripts," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 850–865.

[7] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi, "The chain of implicit trust: An analysis of the web third-party resources loading," in *Proceedings of the international conference on World wide web*, 2019, pp. 2851–2857.

[8] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld, "SandTrap: Securing JavaScript-driven trigger-action platforms," in *Proceedings of the USENIX Security Symposium*, 2021, pp. 2899–2916.

[9] L. Ingram and M. Walfish, "TreeHouse: JavaScript sandboxes to help web developers help themselves," in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 153–164.

[10] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns, "ScriptProtect: mitigating unsafe third-party JavaScript practices," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2019, pp. 391–402.

[11] X. Dong, M. Tran, Z. Liang, and X. Jiang, "AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements," in *Proceedings of the Annual Computer Security Applications Conference*, 2011, pp. 297–306.

[12] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 481–496.

[13] "codesandbox," https://codesandbox.io/.

[14] "playcode," https://playcode.io/.

[15] "Tripadvisor," https://www.tripadvisor.com/.

[16] "Embark," https://github.com/embarklabs/embark.

[17] "Screeps: MMO sandbox game for programmers," https://screeps.com/.

[18] "Algolia," https://www.algolia.com/.

[19] "box.js," https://github.com/CapacitorSet/box-js.

[20] "ses - npm," https://www.npmjs.com/package/ses.

[21] "isolation - npm," https://www.npmjs.com/package/isolated-vm.

[22] "vm2 - npm," https://www.npmjs.com/package/vm2.

[23] "vm2-security-issue," https://github.com/patriksimek/vm2/security.

[24] "isolated-vm-security-issue," https://github.com/laverdet/isolated-vm/security.

[25] "ses-security-issue," https://github.com/endojs/endo/security.

[26] A. AlHamdan and C.-A. Staicu, "SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes," in *Proceedings of the USENIX Security Symposium*, 2023.

[27] "Metamask wallet statistics 2026: Shocking growth," https://coinlaw.io/metamask-wallet-statistics/.

[28] "ses," https://www.npmjs.com/package/ses.

[29] "isolated-vm – access to multiple isolates in nodejs," https://github.com/laverdet/isolated-vm.

[30] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "BreakApp: Automated, flexible application compartmentalization," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[31] "JS-Interpreter," https://github.com/NeilFraser/JS-Interpreter.

[32] "jailed," https://github.com/asvd/jailed.

[33] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan, "AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements," in *Proceedings of the USENIX Security Symposium*, 2010, pp. 371–388.

[34] "vm2," https://github.com/patriksimek/vm2.

[35] "Scope," https://developer.mozilla.org/en-US/docs/Glossary/Scope.

[36] "Standard built-in objects," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects.

[37] "Inheritance and the prototype chain," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[38] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent Spring: Prototype pollution leads to remote code execution in Node.js," in *Proceedings of the USENIX Security Symposium*, 2023.

[39] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2021, pp. 268–279.

[40] Z. Kang, M. Lyu, Z. Liu, J. Yu, R. Fan, S. Li, and Y. Cao, "Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2025.

[41] K. A. Zhengyu Liu and Y. Cao, "Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in Node.js template engines for malicious consequences," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2024.

[42] S. L. Zifeng Kang and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites," in *Proceedings of the Network and Distributed System Security Symposium*, 2022.

[43] "constructor," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor.

[44] "Google caja," https://developers.google.com/caja.

[45] "Global object," https://developer.mozilla.org/en-US/docs/Glossary/Global_object.

[46] "Object.getownpropertynames()," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames.

[47] "typeof," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof.

[48] "Primitive," https://developer.mozilla.org/en-US/docs/Glossary/Primitive.

[49] "Typeerror," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError.

[50] "Error," https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Error.

[51] "js-vuln-db," https://github.com/tunz/js-vuln-db.

[52] "Cve-2022-36067," https://nvd.nist.gov/vuln/detail/CVE-2022-36067.

[53] "Cve-2022-25893," https://nvd.nist.gov/vuln/detail/CVE-2022-25893.

[54] "Cve-2021-23543," https://nvd.nist.gov/vuln/detail/CVE-2021-23543.

[55] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety:type-based verification of JavaScript sandboxing," in *Proceedings of the USENIX Security Symposium*, 2011.

[56] "TreeHouse - default policy," https://github.com/lawnsea/TreeHouse/blob/8c9ad1b54727ce9cb84700c311808990c663506b/src/policy/default.js.

[57] "Caja - policy," https://github.com/googlearchive/caja/blob/master/src/com/google/caja/lang/html/html5-elements-whitelist.json.

[58] "ECMA-262, 13th edition, june 2022 ECMAScript 2022 language specification," https://262.ecma-international.org/13.0/.

[59] "Cross site scripting (XSS) vulnerability payload list," https://github.com/payloadbox/xss-payload-list.

[60] "ECMA-262-3 in detail. chapter 4. scope chain," http://dmitrysoshnikov.com/ecmascript/chapter-4-scope-chain/.

[61] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 2613–2630.

[62] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript engines with aspect-preserving mutation," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020, pp. 1629–1642.

[63] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proceedings of the European Symposium on Research in Computer Security*, 2016, pp. 581–601.

[64] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[65] C. Holler, K. Herzig, A. Zeller *et al.*, "Fuzzing with code fragments," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 445–458.

[66] "Vm2 issue 175," https://github.com/patriksimek/vm2/issues/175.

[67] "Vm2 issue 185," https://github.com/patriksimek/vm2/issues/185.

[68] "Vm2 issue 225," https://github.com/patriksimek/vm2/issues/225.

[69] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6, pp. 253–267, 1999.

[70] "Playwright enables reliable end-to-end testing for modern web apps," https://playwright.dev/.

[71] "Regression test for vm2," https://github.com/patriksimek/vm2/tree/master/test.

[72] "Js delta," https://github.com/wala/jsdelta.

[73] "ECMAScript 2026 language specification - the with statement," https://tc39.es/ecma262/#sec-with-statement.

[74] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," 2014, pp. 861–875.

[75] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[76] "SandDriller - a fully-automated approach for testing language-based javascript sandboxes," https://github.com/vdata1/SandDriller.

[77] "async function," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

[78] D. Klein and M. Johns, "Parse me, baby, one more time: Bypassing html sanitizer via parsing differentials," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2024, pp. 203–221.

[79] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee, "FuzzOrigin: Detecting UXSS vulnerabilities in browsers through origin fuzzing," in *Proceedings of the USENIX Security Symposium*, 2022, pp. 1008–1023.

[80] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases," in *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[81] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler," in *Proceedings of the USENIX Security Symposium*, 2023.

[82] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[83] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[84] "notevil," https://www.npmjs.com/package/notevil.

[85] "realms-shim," https://www.npmjs.com/package/realms-shim.

[86] W. Luo, X. Ding, P. Wu, X. Zhang, Q. Shen, and Z. Wu, "ScriptChecker: To tame third-party script execution with task capabilities," in *Proceedings of the Network and Distributed System Security Symposium*, 2022.

[87] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "WebJail: least-privilege integration of third-party components in web mashups," in *Proceedings of the Annual Computer Security Applications Conference*, 2011, pp. 307–316.

[88] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "NatiSand: Native code sandboxing for JavaScript runtimes," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, 2023.

[89] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: a virtualized browser to sandbox third-party JavaScripts with enhanced security," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2010, pp. 8–9.

[90] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009, pp. 79–93.

[91] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama, "SMash: secure component model for cross-domain mashups on unmodified browsers," in *Proceedings of the international conference on World wide web*, 2008, pp. 535–544.

[92] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011, pp. 363–378.

[93] "Making javascript safe for advertising," https://www.crockford.com/adsafe/.

[94] S. Wi, T. T. Nguyen, J. Kim, B. Stock, and S. Son, "DiffCSP: Finding browser bugs in Content Security Policy enforcement through differential testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2023.

[95] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "SecBench.js: An executable security benchmark suite for server-side JavaScript," in *Proceedings of the International Conference on Software Engineering*, 2023.

[96] C. W. M. P. F. T. Liam Wachter, Julian Gremminger, "Dumpling: Fine-grained differential javascript engine fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2025.

[97] "National vulnerability database (NVD)," https://nvd.nist.gov/.

[98] "Synk vulnerability DB," https://security.snyk.io/.

Appendix A. Studied Vulnerabilities

To analyze the common patterns of sandbox-escaping bugs, we gathered 67 bug reports, including 18 CVEs, and JS tests patching these bugs from NVD [97], Snyk vulnerability DB [98], and eight GitHub projects. We conducted a keyword search with the terms “sandbox escaping" and “JS sandbox breaking" on Google and GitHub Search services. We also referenced the CVE, security advisory, and the issue of nine bugs reported by SandDriller's authors.

We provide a complete list of the studied 67 vulnerabilities, along with our observations, in Table 7. Our analysis revealed two key patterns:

Observation 1. We observed that 74% (49 out of 67) of the sandbox bugs stemmed from guest code gaining access to built-in object references, ultimately allowing the guest to bypass sandbox restrictions.

Observation 2. The remaining 26% (18 out of 67) of vulnerabilities were caused by guest code leveraging allowed interfaces to run arbitrary code in the host environment.

Appendix B. Details of the Found Bugs

Table 8 describes the details of all bugs and misuse cases found by SandVenture. The fourth column presents the details of the 85 bugs and nine misuse cases, with each bug separated by “ / ".

Appendix C. Object Coverage Comparison

We compare SandVenture with SandDriller in terms of object coverage. For each accessed object, we record its `toString` tag, name, and prototype chain. For `Error` objects, we additionally log a message capturing the object or function that triggered the exception. We then compute the number of unique tuples of the form (`tag`, `name`, `chain`, (+`log`)).

We ran SandVenture and SandDriller for 24 hours using the same corpus described in Section 5.4. For this comparison, we evaluate SandVenture with only Strategies 1, 2, and 3, which are the strategies most directly comparable to SandDriller. SandVenture covers 15,608 tuples whereas SandDriller covers only 2,988 tuples. Individually, Strategies 1, 2, and 3 cover 6,851, 6,893, and 3,753 tuples, respectively. These results show that SandVenture explores a substantially broader object space than SandDriller, which increases its opportunity to expose sandbox vulnerabilities.

Appendix D. Mutation Syntax Errors

SandVenture performs AST-based mutation to minimize syntax errors. However, JS imposes context-sensitive semantic constraints that cannot be fully eliminated by AST-level mutation alone. For example, invoking `super()` outside of a class context results in a syntax error.

On average, SandVenture generated 9M inputs in a 24-hour run, of which 114K (1.26%) triggered syntax errors. These errors stemmed from invalid variable or function scoping introduced by (1) wrapping (102K errors) and (2) mutations (12K errors).

Appendix E. Ethical Considerations

We conducted experiments on open-source JS sandboxes, and all experiments were carried out in a local machine. SandVenture found 85 previously unreported sandbox-escaping vulnerabilities. Recognizing the criticality of these findings and the widespread use of JS sandboxes, we responsibly disclosed all vulnerabilities to the corresponding vendors in a timely manner. At the time of writing, all affected vendors had at least 90 days to address the reported vulnerabilities.

We acknowledge the potential risks of releasing SandVenture, as it could be misused by malicious actors. Nevertheless, we argue that security through obscurity is not a sustainable approach to ensuring the security of JS sandboxes. By releasing SandVenture, we aim to support vendors and researchers in hardening JS sandbox frameworks and ultimately reduce the attack surface available to adversaries.

Appendix F. Open Science

We make our work's artifacts (including experiment code and data) available in https://github.com/WebSec-Lab/SandVenture.

Idx	Observation	Sandbox	Bug Identifier
1	**Observation 1:** Referencing Diverse Built-in Objects	Google Caja	SecurityAdvisory20190606
2		Google Caja	SecurityAdvisory201308013-1
3		Google Caja	SecurityAdvisory201308013-2
4		Google Caja	SecurityAdvisory20130502-1
5		Google Caja	SecurityAdvisory20130502-2
6		Google Caja	SecurityAdvisory20130423-1
7		Google Caja	SecurityAdvisory20130423-2
8		Google Caja	SecurityAdvisory20130410
9		Google Caja	SecurityAdvisory20130213-1
10		Google Caja	SecurityAdvisory20120919
11		Google Caja	SecurityAdvisory20090220
12		vm2	CVE-2019-10761
13		vm2	CVE-2021-23449
14		vm2	CVE-2021-23555
15		vm2	CVE-2021-21413
16		vm2	CVE-2022-36067
17		vm2	CVE-2022-25893
18		vm2	CVE-2023-37903
19		vm2	CVE-2023-37466
20		vm2	CVE-2023-32314
21		vm2	CVE-2023-30547
22		vm2	CVE-2023-29199
23		vm2	Issue 138
24		vm2	Issue 175
25		vm2	Issue 184
26		vm2	Issue 185
27		vm2	Issue 186
28		vm2	Issue 187
29		vm2	Issue 199
30		vm2	Issue 224
31		vm2	Issue 225
32		vm2	Issue 241
33		vm2	Issue 268
34		vm2	Issue 276
35		JS-Interpreter	Issue 181-1
36		JS-Interpreter	Issue 181-2
37		safe-eval	CVE-2022-25904
38		safe-eval	CVE-2020-7710
39		safe-eval	CVE-2017-16088
40		safe-eval	Issue 7
41		safe-eval	Issue 16
42		safe-eval	Issue 18
43		safe-eval	Issue 24
44		safe-eval	Issue 26
45		realms-shim	CVE-2021-235940
46		realms-shim	CVE-2021-23543
47		Sandtrap	GHSA-xx7r-mw56-3q2h
48		jailed	CVE-2022-23923
49		notevil	CVE-2021-23771
50	**Observation 2:** Injecting JS Code via Host Object	Google Caja	SecurityAdvisory20171114
51		Google Caja	SecurityAdvisory20180402
52		Google Caja	SecurityAdvisory20160601
53		Google Caja	SecurityAdvisory20160421
54		Google Caja	SecurityAdvisory20160128
55		Google Caja	SecurityAdvisory20131121
56		Google Caja	SecurityAdvisory20130213-2
57		Google Caja	SecurityAdvisory20130122-2
58		Google Caja	SecurityAdvisory20130122-3
59		Google Caja	SecurityAdvisory20120502
60		Google Caja	SecurityAdvisory20121108-2
61		Google Caja	SecurityAdvisory20120116
62		Google Caja	SecurityAdvisory20110802
63		Google Caja	SecurityAdvisory19Oct2009
64		vm2	Issue 177
65		vm2	Issue 179
66		ADsafe	Figure 10 in [55]
67		ADsafe	Figure 11 in [55]

TABLE 7: The details of the studied vulnerabilities along with our observations.

Bug Idx	Strategy	Sandbox	†Vulnerability Details	# of Bugs (or Misuses)
1–4	**Strategy 1**: Referencing built-in objects	safe-eval	- Access to `import` / `this` / `Proxy(this, this)` / `isPrototypeOf`	4
5			- Access to `constructor` (after deleting it)	1
6–9		jailed	- Access to `import` / `this` / `Proxy(this, this)` / `constructor`	4
10		SandTrap	- Access to `this` from the `eval` context	1
11–14		sandbox	- Access to `import` / `this` / `Proxy(this, this)` / `constructor`	4
15–18		near-membrane	- Access to `import` / `this` / `Proxy(this, this)` / `constructor`	4
19	**Strategy 2**: Accessing error objects	safe-eval	- Recursion with `Error` stack calling (`RangeError`)	1
20–21			- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2
22–25			`__lookupGetter__()` / `__lookupSetter__()` / `toString()` / `valueOf()`	4
26–27			`hasOwnProperty()` / `propertyIsEnumerable()` (`TypeError`)	2
28		jailed	- Recursion with `Error` stack calling (`RangeError`)	1
29–30			- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2
31–33			`__lookupGetter__()` / `__lookupSetter__()` / `toString()`	3
34–37			`hasOwnProperty()` / `isPrototypeOf()` / `propertyIsEnumerable()` / `valueOf()`	4
38–41			`setTimeout()` / `clearTimeout()` / `setInterval()` / `clearInterval()` (`TypeError`)	4
42		SandTrap	- Recursion with `Error` stack calling (`RangeError`)	1
43			- Error thrown by assigning a value to `eval.toString` (`ReferenceError`)	1
44–45			`Object.property.set` / `Object.property.get` (`TypeError`)	2
46			- Error thrown by deleting `Array.prototype[Symbol.iterator]` (`TypeError`)	1
47–48		sandbox	- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2
49–52			`__lookupGetter__()` / `__lookupSetter__()` / `toString()`/ `valueOf()`	4
53–55			`hasOwnProperty()` / `isPrototypeOf()` / `propertyIsEnumerable()` (`TypeError`)	3
56		near-membrane	- Recursion with `Error` stack calling (`RangeError`)	1
57–58			- Error thrown by calling `__defineGetter__()` / `__defineSetter__()`	2
59–62			`__lookupGetter__()` / `__lookupSetter__()` / `toString()`/ `valueOf()`	4
63–65			`hasOwnProperty()` / `isPrototypeOf()` / `propertyIsEnumerable()` (`TypeError`)	3
66	**Strategy 3**: Accessing run-time monitoring objects	safe-eval	- Error thrown in the `global` context	1
67–69			- Error thrown in the `async` / `promise` / `WebAssembly` context	3
70		jailed	- Error thrown in the `promise` context	1
71–73		vm2	- Error thrown in the `async` / `promise` / `WebAssembly` context	3
74		sandbox	- Error thrown in the `global` context	1
75–76			- Error thrown in the `async` / `promise` context	2
77		near-membrane	- Error thrown in the `global` context	1
78			- Error thrown in the `async` context	1
79	**Strategy 4**: Injecting HTML elements	TreeHouse	- JS URL containing uppercase letters (i.e., `JaVaScript:`)	1
80–82	**Strategy 5**: Triggering scope chain	ADsafe	- Abuse by scope chain `let` / `const` / `class`	3
83–85		SES	- Abuse by scope chain `let` / `const` / `class`	3
86–88	**Strategy 6**: Invoking policy-allowed APIs	jailed	- Abuse allowed functions for dynamic code conversion / dynamic page redirection / module import	3
89–91		JS-Interpreter	- Abuse allowed functions for dynamic code conversion / dynamic page redirection / module import	3
92–93		SES	- Abuse allowed functions for dynamic code conversion / dynamic page redirection / module import	3
			Total	85 bugs (+9 misuses)

Counted as vulnerabilities Counted as misuse cases † Each bug is separated by " / "

TABLE 8: The details of the bugs that SandVenture found.