

TypePatrol: Adversarial Testing for Uncovering Security-Relevant Type Inconsistencies in JavaScript Libraries

Sai Mallikarjun Galipelli ✉ 

IIT Kharagpur, India

Cristian-Alexandru Staicu ✉ 

Endor Labs, Oradea, Romania

Jibesh Patra ✉ 

IIT Kharagpur, India

Abstract

Background. Gradual typing, as adopted by languages like TypeScript, aims to combine the flexibility of dynamic typing with the safety guarantees of static type systems. However, the deployed gradual typing systems are inherently unsound, meaning in practice, type inconsistencies are common and can hide or cause subtle bugs. This problem is particularly concerning for widely used libraries, where type annotations serve as both documentation and contracts for users. A value with a wrong type in such libraries may trigger cascading failures in downstream code, including unintended control-flow transfers. Existing work has explored testing-based approaches to detect type mismatches. However, these methods fall short in adversarial settings where malicious inputs are intentionally crafted to exploit type inconsistencies.

Aim. This work introduces TypePatrol, the first adversarial testing framework to uncover security-relevant type mismatches.

Method. TypePatrol systematically mutates values in unit tests and observes the runtime types. Our approach focuses on three classes of security-relevant effects: bypassing input sanitization, invoking incorrect methods, and functions returning wrong types.

Results. We apply TypePatrol to 30 popular JavaScript and TypeScript libraries, and discover hundreds of relevant type inconsistencies. In seven cases, we propose fixes through ten pull requests, the majority of which were accepted, underscoring the real-world relevance of our approach.

Conclusions. Crucially, TypePatrol shows that adversarial testing exposes bugs that slip through conventional testing in even the most popular well-maintained projects. We also discuss cases in which type inconsistencies lead to serious security issues, confirming that these bugs are a serious threat to modern software.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories

Keywords and phrases Type Inconsistencies, Dynamic Types, JavaScript, TypeScript, Adversarial Testing, Fuzzing, Software Security

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.17

Category Technical Track Paper

Supplementary Material *Software (Source Code):* <https://github.com/SAFE-Code-IITKGP/TypePatrol>, archived at `swb:1:dir:f79556d46d94713373294d57b6ce42114e2265e1`



© Sai Mallikarjun Galipelli, Cristian-Alexandru Staicu, and Jibesh Patra;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 17; pp. 17:1–17:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Gradual typing [36] has been widely adopted by dynamically typed languages such as JavaScript [37] and Python [24] because it supports rapid prototyping while, at the same time, providing some of the safety guarantees of static typing. TypeScript - a superset of JavaScript, has evolved as a major language that supports gradual typing and integrates smoothly with vanilla JavaScript. Popular projects like the Angular client-side framework or the Jest testing framework use TypeScript both to document their features and to catch bugs early in the development process [13]. The former benefit has led to powerful IDE suggestion capabilities fueled by community-provided types for almost 10K open-source libraries [11].

Early gradual type systems proposed for scripting languages, such as those by Rastogi et al. [32] and Richards et al. [33], are sound in the sense that they ensure that executions do not violate annotated types. Existing gradual type implementations, such as Microsoft's TypeScript compiler, differ significantly from these early proposals by adopting a pragmatic type-checking discipline that performs lightweight static checking with no residual runtime checks. As shown by Feldthaus et al. [12], this, in turn, results in inconsistencies between type hints and the types observed at runtime. More recently, Troppmann et al. [37] show that such mismatches can be exploited by attackers to cause unwanted security effects, as illustrated by security advisories in popular libraries like CVE-2021-23445, CVE-2021-23434. However, their analysis is fully static, and it does not support dynamic analysis of libraries, which is often needed to handle hard-to-analyze features of scripting languages [1].

Let us consider an example that uses the popular JavaScript library *cheerio* (30K GitHub stars and 14M+ weekly downloads) to show how type inconsistencies at a library's API level can lead to unwanted security incidents.

```
import { isHtml } from 'cheerio/utils';

const dangerousInput = ['<img src=x onerror=alert(1)>'];
if (!isHtml(dangerousInput))
  document.write(dangerousInput);
```

Developers intend to restrict user input to HTML content only, which is sometimes considered safe to write directly into the DOM. When provided with a string value, the `isHtml` method behaves as expected. However, when given an input of an incorrect type (e.g., an array), the method returns an erroneous result, potentially leading to a cross-site scripting (XSS) vulnerability. The underlying assumption made by the library developers is that the input will always be a string, as indicated by the inline type hint present in the original TypeScript code `export function isHtml(str: string): boolean`

However, there is no runtime check to enforce this expectation, neither manually added by the developers nor automatically produced by the compiler. While Troppmann et al. [37] speculate about the existence of such issues, their static analysis does not consider library-level type inconsistencies, nor can it reason about the multitude of ways in which functions can be exported in JavaScript libraries. In this work, we argue that testing can be applied to detect the potential for library misuse. Our approach, which we discuss below, discovered this bug automatically, and we proceeded to create the following patch, which was subsequently merged into the codebase by the maintainer¹:

```
export function isHtml(str: string): boolean {
  if (typeof str !== 'string') { // added by our patch
    return false;
  }
  const tagStart = str.indexOf('<');
  // ... unchanged }
```

¹ <https://github.com/cheeriojs/cheerio/pull/4523>

In contrast, existing work on finding gradual type mismatches relies on the completeness of test suites [14] or on type information collected at runtime [12] along typical execution paths. Thus, these approaches are unable to detect subtle bugs like the one above, where *type violations only manifest when an unexpected value is passed by an attacker under adversarial conditions*.

In this paper, we present TypePatrol, an approach to automatically identify gradual type violations. The key insight of our approach is that instead of relying on test suites alone, which often exercise well-behaved paths and values, we mutate values in existing tests to synthesize new tests, which are then used to probe for type violations that may not be triggered during standard executions. In other words, we propose a domain-specific test amplification strategy [8, 9, 4] that aims to increase the likelihood of type mismatches. We argue that this approach mimics how attackers might attempt to compromise library code by passing values of the wrong type. Beyond unwanted effects on input validation as in the above bug, we also discuss how attackers might direct control flow to unintended methods or force the library to produce values of unintended types, potentially enabling downstream attacks. This problem is exacerbated by the extensive transitive use of libraries within the JavaScript ecosystem [41].

We apply TypePatrol to 30 popular JavaScript and TypeScript libraries and show that developers do not strictly enforce annotated types, allowing for potential misuse via type inconsistencies. In our evaluation, TypePatrol detects type mismatches at hundreds of unique code locations in the analyzed libraries. Based on these findings, we propose ten fixes to address adversarial manipulation risks, nine of which were accepted and integrated by the respective maintainers. These results show that our approach identifies relevant, realistic bugs in well-tested, production-grade projects. Moreover, on the duration of this project, one of the authors discovered CVE-2026-25049, a critical vulnerability in n8n, a popular open-source project with 188K GitHub stars. This very serious security vulnerability shows that type mismatches are not merely a theoretical concept, but a very relevant pressing problem in real-world software.

In summary, the contributions of this paper are:

- We define the problem of testing gradually typed code under adversarial conditions and discuss several unwanted security effects that attackers can trigger due to the unsoundness of the type checker.
- TypePatrol: a novel testing technique for finding security-relevant type mismatches in popular JavaScript and TypeScript libraries by mutating the types of values in existing tests.
- An empirical evaluation involving highly popular real-world JavaScript and TypeScript libraries, in which TypePatrol finds hundreds of potentially security-relevant inconsistencies, some of which we help remediate in the associated codebases.

2 Security Considerations

Threat Model. We consider an adversary who can indirectly invoke JavaScript library code (e.g., by carefully crafting inputs to a web application) and can also manipulate the values and types passed into these libraries. We also assume that the attacker has access to the client code but cannot directly modify it. Additionally, all interactions between the library and the attacker happen through publicly exposed APIs. The adversary's goal is to pass values of the wrong types to alter the intended behavior of the library code. Below, we discuss three types of unwanted security effects that an attacker can trigger via type inconsistencies.

17:4 TypePatrol: Adversarial Testing for JavaScript Libraries

Bypass Input Sanitization

Due to automatic type coercions, attackers can manipulate strict equality checks via values with wrong types. Let us consider the following example:

```
if (username === "admin") {  
    alert("You cannot change the admin password!")  
} else {  
    passwords[username] = newVal;  
}
```

In this example, developers prevent the modification of the admin password via the strict equality check in the first line. However, if attackers provide an array value of the type `["admin"]` the equality check will return false due to the type mismatch, but the property access in the else branch will modify the admin password via a type coercion to string. In this way, the attacker can bypass the explicit validation added by developers. Builtin methods like `indexOf` that rely on equality checks in their native implementation can also be abused in this way.

Manipulate Control Flow

Similarly, when invoking methods on values whose type was not explicitly asserted, an attacker can divert the control flow to a method that was not intended by developers. Let us take the following example: `let atLocation = input.indexOf("@");` In this example, the developers intended to retrieve the position of the at symbol, but when presented with an array input, the interpreter will invoke the `Array.prototype.indexOf` method instead of `String.prototype.indexOf`. We call such methods with identical names on different JavaScript built-ins *confusables*.

Alter the Library's Output

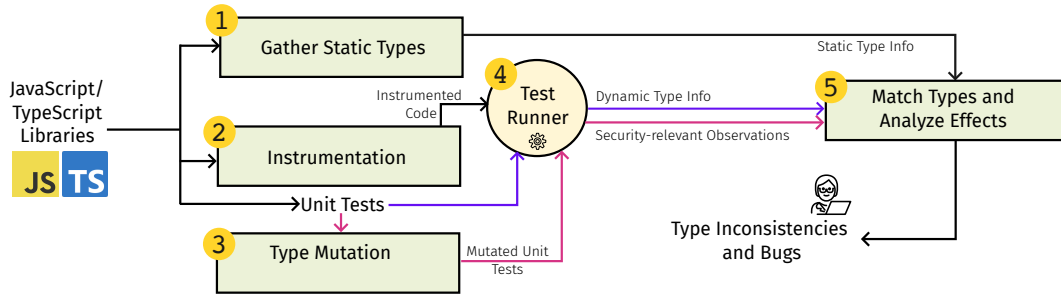
Finally, each method with an annotated return value, whose type mismatches at runtime, is firstly a clear violation of the contract between the library creator and consumer. Moreover, it can be used as a gadget to chain downstream type mismatch compromises. For example, consider the slightly modified example from the introduction:

```
import { isHtml } from 'cheerio/utils';  
import { capitalize } from 'custom-pkg';  
if (typeof input !== string)  
    throw "Wrong input type";  
input = capitalize(input);  
if (!isHtml(input))  
    document.write(input);
```

Let us assume the `capitalize` method is a third-party library that claims to receive a string and output an altered version of it. Even though the developer checked that the input is a string to match this expectation, if the `capitalize` method misbehaves in some cases and produces an array instead of a string, the `isHtml` method can be compromised similarly as before, using the `capitalize` mismatch as a building block for the final exploit.

3 Approach

We follow a five-step approach in TypePatrol to check for type mismatches between developer annotations and the actual types assigned at runtime. Figure 1 shows an overview of our approach where each circled number 1 represent the steps.



■ **Figure 1** The first step of TypePatrol is to gather the static types annotated by developers in the respective JavaScript or TypeScript libraries. Next, we instrument the code followed by mutation of the unit tests. In the fourth step, the test runner executes the original unit tests and the mutated unit tests using the instrumented files. Finally, we match the types obtained from the runtime to the statically obtained types and also analyze the effects of security-relevant observations.

3.1 Gather Static Types

Before conducting adversarial testing of libraries, we want to verify if libraries inherently contain some type inconsistencies i.e., discrepancies between developer-intended types and the actual types observed at runtime. To this end, the first step of TypePatrol involves collecting statically annotated types. Specifically, this phase gathers developer annotated types either directly from the source files or from the corresponding type definition files associated with the library.

For TypeScript libraries, developers typically write the type annotations directly in the source code. In contrast, JavaScript libraries do not contain any type annotations in the source code. Instead, the annotations are mentioned in *.d.ts* files, also known as type declaration file. We define *static type information* in TypePatrol as follows:

► **Definition 1** (Static Type Information). *For a given program P , the static type information of an identifier i , is a triple (i_{τ_s}, P, l) where τ_s is a type annotation by developer and l is the location in P where the type information is observed.*

3.2 Instrumentation

After gathering static type information, the next step in TypePatrol is to observe type information at runtime. This is achieved by instrumenting specific code locations within the target libraries. A primary objective of TypePatrol is to identify potential security-relevant violations that are triggered when values of unintended types are mistakenly propagated. In TypePatrol, we focus on three classes of security-relevant violations: 1) bypassing input sanitization, 2) invoking incorrect methods, and 3) functions returning wrong types. To observe the effects at runtime, we instrument two categories of program locations:

1. *Annotated points*: The program points where developers explicitly provide type annotations. The motivation to collect such locations is to capture the expected types specified by developers. We achieve this by instrumenting the following locations in a program:
 - Function parameters and the lexical redefinitions of function parameters.
 - Property assignments in the constructors of class definitions.
2. *Security-relevant points*: The program locations that are likely to have security implications. Such instrumentation points fall into three main categories. First, locations where developers perform additional checks, such as explicit validation of values. Second, locations where developers invoke commonly used methods. Third, where developers explicitly annotate the type returned by a function.

As discussed in Section 2, we consider these three categories of program points to be potentially security-relevant because, if the checks can be bypassed or manipulated by an adversary, or if an adversary can arbitrarily invoke a method, the program may return unintended values, may suffer from ineffective input validation, inconsistent states, type confusion, or other exploitable runtime errors. To this end, we keep track of the following locations of a program:

- Input sanitization by developers to ensure the expected value. To keep the overall runtime of the instrumented code manageable, we only focus on cases where the developers check a value with another string literal. Since string type in JavaScript shares many common methods with other types, prior work [37] has shown that developers are likely to encounter type confusions for such validations (Section 2).
- Method calls involving common function name. In JavaScript, many types share method names. We consider such cases as security-relevant because an attacker can exploit developers type confusion to induce control flow to a method that is valid for a type but unintended by the developer (Section 2). In particular, we instrument the calls involving the following confusable method names that are shared by multiple types: `indexOf`, `includes`, `lastIndexOf`, `toString`, `concat`, `slice`, `forEach`, `toLocaleString`, `valueOf`.
- Return statements in a function. Although return statements are typically explicitly type annotated by developers, we consider them as security-relevant since an adversary can potentially alter behavior of a library through these points. Section 2 contains one such example to demonstrate this.

Consider the following JavaScript code as an example. The code denotes two categories of program instrumentation points. The `•` in the code shows the instrumentation locations for annotation points while the `*` shows the security-relevant points.

The code denotes the instrumentation points to keep track of the function parameters, lexical redefinitions of a parameter, invocation of a common method, input value sanitization and the return statement. As a concrete example, the code contains a `calc` function that contains one parameter `inp` as input marked as `•` showing a location that is typically explicitly annotated by developers. Line 8 of the code invokes a method `includes` which is a security-relevant instrumentation point marked as `*`. Similarly, line 10 validates the input to a string literal which is marked with `*` to signify its security relevance.

```

1  class Rectangle {
2      constructor(height, width) { •
3          this.height = height; •
4          this.width = width; •
5      }
6      calc(inp) { •
7          let area = 0;
8          let check_intent = inp.includes("area"); *
9          if (check_intent) {
10             if (inp === "double_area") *
11                 inp = 2; •
12             else
13                 inp = 1; •
14             area = (this.height * this.width) * inp;
15         }
16         return area; *
17     }
18 }

```

Algorithm 1 Mutate Unit Test.

Input: Unit test U with parameters $X = [x_1, \dots, x_n]$ Output: Mutated parameters $X' = [x'_1, \dots, x'_n]$ 1: $X' \leftarrow []$ 2: for each $x \in X$ do 3: $\tau \leftarrow \text{TYPEOF}(x)$ 4: $n \leftarrow \text{RAND}(\mathbb{N})$ 5: $s \leftarrow \text{RAND}(\mathbb{S})$ 6: $b \leftarrow \text{RAND}(\mathbb{B})$ 7: if $\tau = \text{string}$ then	8: $x' \leftarrow \text{RAND}(\{n, b, [x]\})$ 9: else if $\tau = \text{number}$ then 10: $x' \leftarrow \text{RAND}(\{s, b\})$ 11: else if $\tau = \text{boolean}$ then 12: $x' \leftarrow \text{RAND}(\{s, n\})$ 13: else 14: $x' \leftarrow \text{RAND}(\{n, s\})$ 15: end if 16: $X'.\text{APPEND}(x')$ 17: end for 18: return X'
---	--

3.3 Test Mutation

Developers routinely use tests to verify correctness and simulate the typical and expected behavior of code. One of the goals of TypePatrol is to induce atypical behavior and simulate how an adversary might interact with the code. To achieve this, we leverage existing unit tests and execute the code by providing values of unexpected types. By introducing such adversarial inputs, TypePatrol aims to elicit unintended behavior and uncover security-relevant violations, simulating how attackers might attempt to manipulate the target code.

Our approach to introducing unexpected types relies on lightweight mutation of function arguments of unit tests. The mutation strategy primarily targets identifiers and literals, as these are most commonly used as function inputs. Literals such as Array literals and Object literals in JavaScript can be structurally complex and composed of multiple other types, we generally exclude them from mutation. However, in practice, we observe that object literals are frequently used as inputs in unit tests. As an exception, we selectively mutate objects and the object literal properties to primitive types.

Algorithm 1 outlines our approach to mutate an unit test. The input to the algorithm is a list of parameters $[x_1, x_2, \dots, x_n]$ to be mutated. For each parameter, a new value is selected from pre-populated pools of *number*, *string* and *boolean*. The type of the selected value is decided based on the following strategy:

- If the parameter x is of type *number*, it is mutated to either a *string* or a *boolean* with each type chosen with equal probability. Similarly, if x is of type *boolean*, it is mutated to either a *string* or a *number*. (line 10, 12 of Algorithm 1).
- If the parameter x is of type *string*, it is mutated to one of the following: a *number*, another *boolean*, or an *array* with equal probability (line 8 of Algorithm 1).
- Beyond the types *number*, *boolean* and *string* the algorithm always mutates x to either a *number* or a *string* (line 14 of Algorithm 1).

The special treatment of strings is motivated by the observation that the string and array types in JavaScript share certain behaviors (e.g. index access method, length computation), making arrays a meaningful mutation target for strings.

3.4 Test Runner

To capture runtime types, we exercise the instrumented code locations identified in Section 3.2. For this purpose, we leverage developer-provided unit tests. Since unit tests are designed to check the correctness of code, they typically achieve deep coverage of program functionality.

As a result, they are suitable for our purpose of adversarial testing of the libraries. The output of test runner phase is *dynamic type information*.

► **Definition 2** (Dynamic Type Information). *For a given program P , the dynamic type information of an identifier i , is a triple (i_{τ_d}, P, l) where τ_d is a literal type that gets assigned at runtime and l is the location in P from where the type information is observed.*

To contrast between the typical and atypical code execution, the test runner phase executes two sets of unit tests. First, the default unit tests provided by the developer and second the unit tests obtained by mutation as described in Section 3.3. After executing these two sets of tests, the next step is to compare the dynamic type information obtained for each case with the static type information which is described further in the next Section 3.5.

3.5 Match Types and Analyze Effects

The final step of TypePatrol is two fold. First, to compare the runtime types obtained by running the tests in Section 3.4 to their static counterparts. We check if types annotated by developers in the respective JavaScript or TypeScript libraries match with the runtime types. Second, to observe the effects of mutation on the security-relevant points introduced in Section 3.2.

Type Matching

While comparing types, if a mismatch is detected, TypePatrol produces a warning, which we refer to as a *type inconsistency* and define below.

► **Definition 3** (Type Inconsistency). *For a program P , the type of an identifier i is inconsistent if the static type i_{τ_s} is lexically different from the dynamically obtained type i_{τ_d} i.e., $i_{\tau_s} \neq i_{\tau_d}$.*

A type inconsistency warning indicates a potential programming error or, in some cases, a security vulnerability. Our type matching step consists of two key phases:

- *Static vs Runtime Comparison:* We compare the developer-annotated static types with the actual types observed at runtime.
- *Mutation Behavior Comparison:* We compare:
 - the behavior observed when executing the developer-provided unit tests, and
 - the behavior observed when executing the corresponding mutated unit tests.

This comparison aims to find any behavioral differences induced by the mutated tests.

Prior work on dynamically typed languages [29, 31] has shown that many type related warnings are likely false positives. For example, two types can be polymorphic and still be reported as likely incompatible resulting in such false positives. To minimize the impact of such false positives, we constrain the type matching domain for *static vs runtime comparison* to only include primitive types: `number`, `string`, `boolean`, `null`, `undefined`. As a final filtering step, we report only those type inconsistency warnings where at least one of the types involved in the mismatch belongs to this set of primitive types.

Analyze Effects

With the threat model presented in Section 2 in mind, we examine the additional behaviors induced by mutated unit tests at the observed security-relevant instrumentation points (Section 3.2). To this end, we focus on three types of new behavior:

1. Executions of input-value sanitization checks during runtime.
2. Runtime errors indicating wrong method invocations. Such errors occur when a value is used to invoke a method without first checking that it has the expected type. In JavaScript, types such as *string* and *array* share common method names. Without proper type checks in place, adversaries can manipulate value types to redirect execution to methods already present in the source code, thereby reusing code for malicious purposes.
3. New type mismatches in function return values. We consider such mismatches security-relevant because an adversary can change the type of an argument passed to a function and potentially cause the function to return sensitive data or otherwise alter the behavior of a library. We argue that functions should validate their parameters at runtime to avoid such scenarios.

4 Implementation

We use Babel as the primary library for instrumenting JavaScript files, leveraging its parser, traverse, and generator packages to manipulate JavaScript ASTs and inject logging or mutation code. For TypeScript files, we perform instrumentation by using the TypeScript Compiler API (the `typescript` npm package), which enables parsing, traversing, and transforming TypeScript ASTs for similar purposes. We use Node.js as the JavaScript runtime environment for executing, building, and testing all JavaScript and TypeScript packages. For reporting the line coverage, packages such as Istanbul, nyc & C8 are used. All scripts and instrumented code are run using Node.js (version 22.13) on a machine equipped with an Intel Core i5-13500 processor, 16 GB RAM, running Ubuntu 24.04.3 LTS.

5 Experimental Setup

We evaluate the effectiveness of TypePatrol on popular libraries written in JavaScript and TypeScript. To this end, we select npm [27] libraries from a list of most-dependent upon JavaScript [22] and TypeScript [23] packages. Such packages are ranked as the most depended upon, which means they are considered among the most widely used in the npm ecosystem, which in practice translates to thousands of dependent packages and millions of weekly downloads. We observe that the primary source code for the packages is either JavaScript (JS) or TypeScript (TS), with TypeScript being compiled to JavaScript. For each package, we download the source code from its corresponding GitHub repository and conduct our experiments.

To accurately extract developer-provided type annotations, a filtering step is applied to the JavaScript packages. Specifically, if a JavaScript package does not contain a `.d.ts` file as part of its source code, we exclude it from our experiments. A `.d.ts` file, also called type declaration file is typically included as part of the code and contains only type information of different code constructs, thereby aiding static analysis tools. For some packages, where the developers do not choose to include a type declaration file, there are community-led efforts such as *DefinitelyTyped* [7] to provide such a file. We choose to exclude such libraries since they may not faithfully reflect a library developer’s original intent.

The first column of Table 1 lists the libraries selected for evaluation along with their weekly download statistics at the time of writing, while the second column reports the number of dependent packages. In total, we evaluate TypePatrol on 30 libraries: fifteen JavaScript (top section of Table 1) and fifteen TypeScript (bottom section of Table 1).

17:10 TypePatrol: Adversarial Testing for JavaScript Libraries

■ **Table 1** Runtime statistics, type inconsistencies, and adversarial testing results for JavaScript (top 15) and TypeScript (bottom 15) libraries. **Legend:** **Dep** = Dependent packages, **IL** = Instrumented locations, **TE** = Total entries, **UL** = Unique locations covered, **LC** = Line coverage (%), **MM** = Mismatches, **UM** = Unique locations in mismatches, **PM** = Type mismatches involving a primitive, **PU** = Unique locations involving a primitive, **AU** = Unique locations covered during Adversarial Testing (+new locations), **AL** = Line coverage during Adversarial Testing (%), **ER** = Extra return entries, **NF** = "Is not a Function" Error, **IS** = Input value sanitization, **CM** = Confusable method usage.

	Runtime Statistics					Type Inconsistencies				Adversarial Testing					
Library (Downloads)	Dep	IL	TE	UL	LC	MM	UM	PM	PU	AU	AL	ER	NF	IS	CM
JavaScript Libraries															
chalk (384M+)	185k+	165	1,732	111	99.61	1	1	1	1	111	99.62	0	0	0	1
del (12M+)	30k+	32	18,244	29	99.18	13	1	1	1	29	99.18	0	0	0	0
dotenv (85M+)	107k+	67	4,118	67	100	34	2	1	1	67	100	0	30	0	2
eventemitter3 (68M+)	9k+	46	10,814	45	100	0	0	0	0	45	100	0	10	0	0
find-up (184M+)	5k+	17	830	17	100	27	3	13	3	17	100	0	0	0	0
form-data (117M+)	15k+	86	3,362	85	98.08	18	2	3	1	85	98.46	2	10	0	0
globby (72M+)	17k+	120	176,670	118	99.2	32	6	29	5	118	98.67	0	0	0	0
iconv-lite (155M+)	9k+	82	46,673.377	56	88.17	10	5	10	5	57 (+5)↗	88.75	0	4	0	0
log4js-node (6M+)	4k+	688	92,515	660	99.09	136	3	132	2	661 (+1)↗	95.39	0	108	0	3
node-portfinder (9M+)	10k+	57	8,641	52	76.61	0	0	0	0	52	73.92	0	9	0	0
Ora (50M+)	54k+	55	4,678	52	93.89	4	3	3	2	52	94.36	0	14	0	0
postcss (131M+)	197k+	960	2,382,709	930	100	6,518	93	268	33	935 (+13)↗	100	6	20	13	5
query-string (14M+)	5k+	182	13,891	176	98.71	109	9	108	8	176	99.26	0	8	1	2
sharp (32M+)	10k+	437	292,463	412	99.73	4	4	4	4	192 (+2)↗	66.75	0	2	3	0
winston (17M+)	27k+	309	313,087	258	74.5	76	25	10	9	259 (+4)↗	69.78	0	13	0	0
TypeScript Libraries															
async-validator (1.5M+)	5k+	135	3,824	55	88.84	1,060	22	303	11	58 (+3)↗	89.99	1	0	2	0
cheerio (14M+)	27k+	517	170,826	431	99.56	81,852	229	6,523	65	437 (+20)↗	99.72	4	65	0	2
chokidar (104M+)	26k+	220	156,331	176	93.09	46,648	59	40,029	41	9	18.82	0	0	0	0
cookie (97M+)	5k+	49	7,267	49	100	78	4	0	0	49	100	0	42	0	2
got (26M+)	12k+	410	207,808	301	96.05	44,790	72	14,954	22	300	95.93	2	15	8	0
helmet (6M+)	5k+	73	1,789	57	97.46	263	15	4	1	53	97.74	0	47	0	0
http-proxy-mid. (20M+)	9k+	100	3,100	71	89.67	1,059	31	191	6	75 (+11)↗	99.32	2	0	0	1
marked (23M+)	16k+	301	10,958	173	87.02	1,584	19	30	2	175 (+6)↗	88.27	2	58	0	1
mime (90M+)	10k+	18	1,521	4	99.91	6	2	6	2	0	99.91	0	5	0	0
MQTT.js (1M+)	4k+	414	265,821	224	87.78	18,159	40	3,287	10	172 (+7)↗	43.42	0	1	0	0
node-glob (246M+)	69k+	229	1,789	156	85.27	671	57	169	15	158 (+2)↗	85.27	0	14	0	0
node-lru-cache (291M+)	7k+	181	2,162,029	149	99.92	1,724,365	85	1,382,626	61	141	99.46	1	10	0	0
node-tar (54M+)	8k+	482	14,408,401	447	99.58	321,538	106	195,403	72	442	95.41	0	12	0	0
uuid (191M+)	71k+	155	217,142	90	84.17	21,553	33	21,164	20	90	84.17	0	0	0	0
webpack-merge (16M+)	35k+	48	5,876	30	98.58	1,948	10	1,354	5	32 (+8)↗	98.58	3	21	0	1

Table 1 also presents the *Runtime Statistics* collected by running TypePatrol on each selected package. As an illustrative example, consider the JavaScript package *postcss* that has over 197K dependent packages and more than 131M weekly downloads. During the instrumentation phase, TypePatrol identified 960 locations in the code that are either annotation or security-relevant points. Upon executing the unit tests of the *postcss* package, we record a total of 2,382,709 runtime entries. This large number of runtime entries arise because the same instrumentation points are repeatedly executed during running the tests. On further analysis we observe that the runtime entries corresponds to 930 unique locations within the source code. Additionally, the line coverage of the instrumented library is 100%. These results show that the considered packages are fairly large, and their unit tests trigger many relevant locations for TypePatrol with high line coverage.

6 Evaluation

Our evaluation of TypePatrol aims to answer the following research questions:

1. How many type inconsistencies does TypePatrol detect in well-tested libraries?
2. Is adversarial testing of libraries able to uncover any unintended behavior?
3. What is the effect of the time budget during adversarial testing?
4. What is the real-world impact of TypePatrol?

6.1 Detecting Type Inconsistencies in Real-World Libraries

Developers typically annotate types to match runtime expectations. To check whether this holds, we compare statically annotated types with the types observed during unit test execution. The *Type Inconsistencies* part of Table 1 summarizes the results of this experiment, with the JavaScript libraries shown in the top section and the TypeScript libraries shown in the bottom section.

The mismatches (MM) column represents the absolute number of type mismatches found while running the unit tests. The number of mismatches can be high because, during execution, the same location can be executed multiple times, e.g., due to loops. The column unique locations represents how many of the absolute type mismatches correspond to distinct code locations. Consider the *postcss* package where the absolute number of mismatches is 6,518 which corresponds to 93 unique locations. As explained in Section 3.5, one of TypePatrol’s goals is to minimize false positives and provide developers with precise warnings about type mismatches. To achieve this, we restrict our analysis to type mismatches where at least one involved type is a primitive, i.e., to avoid reasoning about type equivalence in complex polymorphic scenario. This result is shown by the last two columns of the *Type Inconsistencies* part of Table 1. For example, for the *postcss* library, we find 268 such type mismatches that involve at least one primitive, of which 33 are unique. The overall results show that type inconsistencies are prevalent in the considered libraries, with at least one mismatch location in each case.

As a concrete type mismatch example found by TypePatrol, consider the following code taken from the *mqtt.js* package that is annotated by the developer to accept an array of string elements and return a string. As seen in line 6, it is possible for the function to return a `null` value. Thus, the correct return type annotation should be `string | null`, which is observed while running the corresponding unit tests. Developers accepted this fix.

```

1 export function validateTopics(topics: string[]): string {
2   if (topics.length === 0) return 'empty_topic_list';
3   for (let i = 0; i < topics.length; i++) {
4     if (!validateTopic(topics[i])) return topics[i];
5   }
6   return null
7 }
```

6.2 Adversarial Testing Outcomes

Section 3.3 explains our approach to mutating unit tests, which simulates how an adversary might induce atypical behavior in code. In this experiment, we examine whether executing mutated unit tests leads to any new observable behavior. For each library in our experimental setup, we mutate one unit test at a time and then rerun the full test suite. To keep the overall approach tractable while still exploring adversarial behavior, we mutate up to 3% of the tests in each library and repeat the experiment for 10 iterations with different random seeds. Although we establish this 3% upper bound, in practice we were often unable to reach it due to assertion and test failures. To validate this choice, we also explored larger mutation budgets. We run experiments with 10%, 25%, and 50% mutation budgets. While these larger budgets increase the total number of runtime entries, they execute fewer tests and do not reveal any additional unique locations in the categories reported under *Adversarial Testing* in Table 1. We therefore focus our evaluation on the 3% mutation budget, which provides a better balance between test-suite execution and adversarial exploration.

The unique locations covered (AU) and line coverage (AL) columns in the *Adversarial Testing* portion of Table 1 report runtime statistics from our adversarial testing phase. We find that, for most libraries in our experimental setup, the mutated unit tests achieve coverage that closely matches that of the original developer-provided unit tests, both in terms of unique locations covered (UL column) and line coverage (LC column). Furthermore, for eight libraries, the adversarial testing phase of TypePatrol exercises additional lines of code, shown as + values in parentheses, that were not covered by the original unit tests. In absolute terms, our approach achieves greater coverage for twelve libraries, denoted by  in the table. For *chokidar*, the mutated tests cover only a small number of locations because many tests fail under mutation. Overall, these coverage results show that adversarial testing preserves much of the original test-suite coverage while exposing additional code in several libraries. The results of Table 1 show that for every library considered in this experiment, there exist inconsistencies between the observed runtime type and the annotated static type provided by the developers. To understand the prevalence of such large number of type inconsistencies, we manually analyze a sample of 30 warnings produced by TypePatrol. We find that there exists two types of such mismatches. First, where one of the type is a primitive but the mismatched type is an user-defined type, i.e., because there might still be legitimate overlap because of duck typing. This kind of mismatches need further investigation to be validated and is beyond the scope of this work. Second, where both types are primitives and considered clear mismatches and need the corresponding developers attention.

6.2.1 Security-relevant Outcomes

When executing the mutated tests, we track the resulting behavior changes. In particular, we analyze the effects of three types of behavior changes to be security-relevant which are explained in Section 3.5. The *Adversarial Testing* part of Table 1 present the results of analyzing the effects of observing 1) return statements (column *ER*) 2) errors caused by incorrect method invocations (column *NF*) and 3) observing input value sanitization by the developer (column *IS*).

6.2.1.1 Observing Return Statements

The *Extra return entries* (ER) column of Table 1 reports the number of times a mutated test input produced additional return entries. Of the 30 packages in our experimental setup, nine exhibit return type mismatches. The prevalence of return type mismatches under mutated unit tests reveals three possible shortcomings in these libraries. *First*, the existing unit tests may be insufficient to trigger this behavior; otherwise, it would likely have been uncovered by the results in Section 6.1. *Second*, developers may make mistakes when writing static type annotations. *Third*, libraries may lack sufficient checks for unexpected argument types, which can cause functions to return unexpected types.

6.2.1.2 Observing Method Calls

As elucidated in Section 3.2, certain types in JavaScript, such as `Array` and `String`, share many common methods; we call these *confusables*. An adversary can exploit this property by passing an incorrect type that still supports the expected method calls, thereby manipulating the control flow. The motivation for this experiment is to evaluate the likelihood of control flow hijacking or manipulation.

■ **Table 2** Top-5 methods involved in errors; three are confusable.

Method	Freq.	Types
<code>indexOf</code>	54	String/Array
<code>replace</code>	43	String
<code>charCodeAt</code>	25	String
<code>slice</code>	22	String/Array
<code>forEach</code>	7	Array/Map/Set

Directly invoking a method on a value of an unexpected type results in an error in most cases. In JavaScript runtimes, such errors typically appear with messages such as "Uncaught `TypeError: is not a function`". The *is not a function error* (NF) column of Table 1 reports these findings by counting the number of such errors encountered. We find that 22 of the 30 packages in our experimental setup exhibit this behavior.

We further analyze the method names that appear in the generated error messages, as well as cases where no error is thrown but a confusable method is invoked. This analysis is based on data collected over ten iterations, with 3% of the possible locations in unit tests mutated in each iteration. The motivation behind this experiment is to determine how often confusable methods are invoked by developers which can be possibly exploited by an adversary. Table 2 lists the top five most common method names and their frequency. For our case, TypePatrol passed a type for which the confusable method does not exist which resulted in the error. Our results show that the majority of these methods are shared across multiple built-in types, making them confusable. For example, the `forEach` method, the fifth most frequent method involved in such errors is common to three types `Array`, `Map`, and `Set`.

The last column (*confusable method usage*) (CM) of Table 1 reports the cases where a confusable method is invoked during adversarial testing but does not result in an error. Our analysis shows that adversarial testing is able to invoke a confusable method for ten libraries where the execution did not cause an error and highlight the importance of proper type sanitization by developers, as the lack of errors may silently result in unintended behavior.

6.2.1.3 Observing Input Value Sanitization

During the adversarial testing of the libraries, we observe the input value sanitization implemented by developers, as described in Section 3.2. Our analysis shows that, in practice, only a small number of libraries exhibit this behavior. For this evaluation, we mutate up to 3% of the unit tests for ten iterations and summarize the results in the column *input value sanitization* (IS) of Table 1. The table shows five out of the 30 libraries in our experimental setup perform such sanitization, with *postcss* exhibiting the highest number of cases. This result highlights the prevalence of possible location for bypassing input sanitization.

As a concrete example of a security-relevant mismatch, consider the package *http-proxy-middleware*, which has close to 20 million weekly downloads². Our analysis reveals the possibility of a remote denial-of-service (DoS) vulnerability if a custom logger is misconfigured. The middleware expects the logger to implement three methods: `info`, `warn`, and `error`. If any of these methods are missing, and an attacker sends input that triggers a code path invoking the absent method, the server will crash.

According to the original type declarations, the following TypeScript code compiles successfully and starts the server, even though the custom logger is incomplete:

² <https://www.npmjs.com/package/http-proxy-middleware>

```

1 import { createProxyMiddleware } from 'http-proxy-middleware';
2 import * as express from 'express';
3 const app = express();
4 const badLogger = { // Broken logger: missing warn/error
5   info: (msg: string) => { console.log('INFO:', msg); } };
6 app.use('/api', createProxyMiddleware({
7   target: 'http://example.com',
8   changeOrigin: true,
9   logger: badLogger,
10 }));
11 app.listen(3000, () => {
12   console.log('Server running on http://localhost:3000');});

```

Here, a custom logger is passed in as one of the options. The internal implementation of `http-proxy-middleware` relies on the `getLogger` function to obtain the logger for downstream usage from the provided options during initialization of a proxy middleware. When a request such as `curl http://localhost:3000/api` is made, the server crashes with a `TypeError` because the middleware attempts to call `logger.error`, which does not exist on `badLogger`. This occurs because the original type definition for the logger option was declared as `Logger | any`. As a result, TypeScript could not detect the missing methods at compile time.

We discover this mismatch during unit test mutations. For example, the following simplified unit test code demonstrates that `getLogger` accepts invalid inputs without validation:

```

1 import { getLogger } from '../src/logger';
2 const logger = getLogger({ logger: 500 });
3 console.log(logger); // prints 500

```

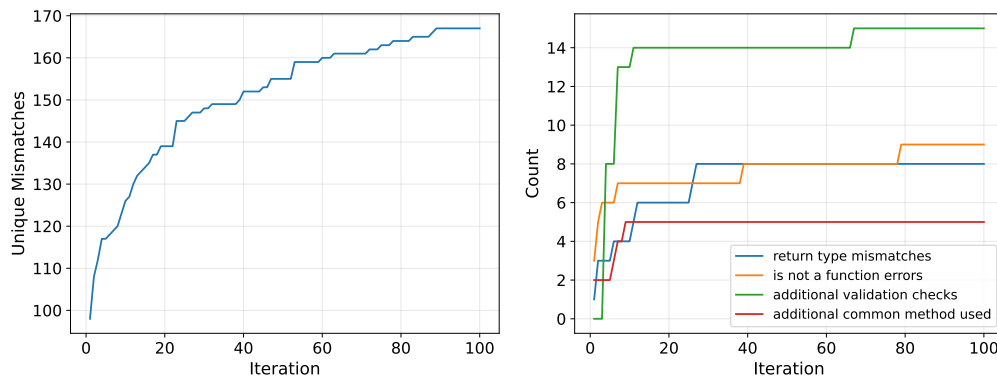
As discussed in Section 3.3, our mutation strategy also targets object literal property values: while the original unit test passed the object `console` as the logger, our mutation replaced it with a string, and the test still executes without error. As discussed in Section 6.2.1, by observing return values at runtime, we detect that the returned value does not conform to the expected `Logger` type. This shows that `getLogger` lacks input validation, leading to a violation of its declared return type. Since `http-proxy-middleware` depends on `getLogger` to retrieve the logger during middleware initialization, this unchecked mismatch exposes the possibility of a remote DoS attack if a misconfigured logger is used and an attacker can trigger a code path that invokes a missing method.

Our recommendation is to add runtime validation to ensure that the provided option is actually a `Logger`. However, the maintainers chose instead to update the type declaration for the logger to accept only `Logger` instead of `Logger | any`. This change ensures that the TypeScript compiler raises an error whenever an incomplete logger is provided.

6.3 Effect of Increased Time Budget during Adversarial Testing

The adversarial testing results of Table 1 show the mismatches found while mutating up to 3% of the unit tests. We adopt this threshold to ensure that optimal number of tests are executed. To further validate these findings, we verify the results of running this experiment on a larger time budget for one randomly selected package. The motivation behind this experiment is to investigate the effect on the number of new mismatches found over time, and to assess the overall impact of the mutations made by TypePatrol.

For this purpose, we mutate 3% of the unit tests in the `postcss` package for 100 iterations. The entire experiment took approximately 40 minutes to complete. In each iteration, a new seed value is used, ensuring that the set of tests selected for mutation varied. The results of this experiment are shown in Figure 2.



■ **Figure 2** Iteration vs Unique Mismatches (left). Iteration vs Kinds of Observed Security-relevant Behavior (right).

The figure on the left presents the cumulative unique mismatches discovered over 100 iterations. A unique mismatch is defined as a specific code location where a developer’s annotated type differs from the dynamically observed type. The plot reveals a linear increase in the number of unique mismatches throughout the experiment’s duration. This demonstrates that, across various mutations of unit tests, TypePatrol is able to effectively explore different parts of the code, continuously uncovering new and unique type mismatches.

The figure on the right shows the different kinds of new behavior observed over 100 iterations. For this experiment, we observe the effects that typically represent possible exploitable locations in code. This result demonstrate that with bigger time budget TypePatrol can uncover new security-relevant behavior in code.

6.4 Real-World Impact

At the time of writing, our evaluation has resulted in total ten pull requests across seven libraries out of the 30 present in our experimental setup. As evident, for some libraries, we find multiple such type reports. Out of the ten pull requests created by us, nine have been acknowledged by the respective developers and successfully merged to the code base. Six of these nine reports by us were merged within two days of report which demonstrates that developers deem such type mismatches important. The overall results and the developers acknowledgment of our reports demonstrate the practical applicability of TypePatrol.

In the following, we present the kinds of changes proposed by us to the developers:

- Adding a runtime validation of type, such as the following code. TypePatrol found that at runtime, passing any other type results in unintended behavior, and as a result, we suggest runtime type validation to the developers. We report two such types of changes, one of which was accepted, while for the other, developers fixed the issue by adding an additional type hint.

```
if (typeof boundary !== 'string')
  throw new TypeError('FormData boundary must be a string');
```

- Improve existing type hints. For example, in the following function, the original code (first line) mentions only string as the return type annotation although the body contains a `return null;` statement. TypePatrol is able to discover this inconsistency and suggest improvements (second line). For all cases, such improved type hints were accepted by developers.

```
function validateTopics(topics: string[]): string
function validateTopics(topics: string[]): string | null
```

7 Threats to Validity and Future Work

Selection of Libraries

The libraries on which TypePatrol is evaluated are based on a ranked list which may change over time. A different selection of JavaScript and TypeScript libraries could lead to different conclusions compared to the current results.

Limitations on the Selection of Instrumentation Points & Types

TypePatrol instruments two categories of points, annotation points and security-relevant points, and our results are based solely on observations at these. Altering this selection, or considering other potentially security-relevant points, could yield different outcomes. We mitigate this by selecting points that developers typically annotate and locations where prior work [37, 24] has shown to encounter bugs. Our reported findings also involve only primitive-type mismatches; considering other types may yield different results.

Unit Test Mutations

For our unit test mutation experiment, we choose to mutate 3% of all unit tests present in each library. These tests are selected randomly, and as a result, executing a different set of mutated unit tests could lead to different observed outcomes. We keep the overall approach reproducible, we follow the standard practice of using a fixed seed during mutation. Additionally, changing mutation threshold of 3% may also affect the results. Finally, the observed behavior during adversarial testing may not always be directly caused by the mutations performed by TypePatrol but, in some cases, due to the non-deterministic code in the underlying library.

Future Work

TypePatrol leaves ample opportunity for future work. The code instrumentation points can be extended to include additional patterns that may be security-relevant. The mutation algorithm used in the adversarial testing phase can also be improved by supporting more data types and crafting more complex input values. TypePatrol filters the observed type inconsistencies by considering only cases where one of the type involved in the mismatch is a primitive type. We plan to improve this by also considering user-defined and polymorphic types. Finally, TypePatrol can integrate an automatic synthesis framework to repair the observed inconsistencies.

8 Related Work

Types in JavaScript

Pradel et al. [29] were the first to propose dynamic analysis for detecting type inconsistencies. However, they do not consider malicious inputs crafted by an attacker, as was recently proposed by Troppmann et al. [37]. David et al. [10] discuss how type inference and stronger type enforcements can prevent deserialization attacks in PHP. Pradel and Sen [30] study the prevalence of type coercions, which are an important culprit for the security issues discussed in this paper, which can be prevented by a stricter type system. Malik et al. [26] advocate for synthesizing type annotations from documentation, while Pradel et al. [28] also advocate for using code context for this goal.

Hoeflich et al. [14] propose synthesizing contracts from type declarations, i.e., runtime checks at the annotation boundary, to enforce gradual types at runtime in JavaScript. Similar to Feldthaus and Möller [12] they find that many type annotations are inconsistent with the types observed at runtime. However, neither of these approaches discuss security-relevant mismatches or assume attackers that can purposely pass values of wrong types.

JavaScript Security

Traditionally there was a lot of work on JavaScript security in the context of client-side web applications, e.g., XSS [20]. However, in recent years there was an increased interest in analyzing the security of open-source libraries, in isolation, mostly in the context of server-side ecosystems like npm [41]. Xiao et al. [39] discuss hidden property attacks, in which adversaries introduce well-crafted properties in complex objects passed to libraries to achieve various unwanted effects. Similarly, Shcherbakov et al. [35] show that a related problem called prototype pollution is prevalent in JavaScript libraries and that it can often be lifted to remote code execution. Kang et al. [16] show that this vulnerability is also prevalent on the client-side where it can be used as a building block for sophisticated attacks like cross-site scripting [17]. Li et al. [25] propose using object dependence graph, a sophisticated way of modeling a program to detect prototype pollution vulnerabilities. Li et al. [21] extend this approach to detecting other taint-style vulnerabilities. Kang et al. [15] propose an enhancement to speed up abstract interpretation in this domain. Differently, Cassel et al. [6] propose using provenance graph for finding vulnerabilities in server-side JavaScript. Finally, Bhuiyan et al. [3] propose a benchmark of known vulnerabilities with exploits, including many prototype pollution entries. While prototype pollution and hidden property abuse can be prevented by enforcing complex types annotated by developers in the code, no prior work tests the prevalence of these vulnerabilities in the presence of type annotations.

JavaScript Testing

There is a plethora of prior work using existing unit tests for dynamic analysis, e.g., [19], or for synthesizing tests automatically. Turcotte et al. [38] and Koishybayev and Kapravelos [18] use existing tests to debloat libraries, i.e., to remove unused parts of the code that might contain vulnerabilities. Selakovic et al. [34] proposes a way of generating tests for higher-order functions, which Artega et al. [2] extends to support nested callbacks. Cassel et al. [5] propose an automatic technique for synthesizing exploits for known vulnerabilities. Finally, Zeng et al. [40] propose a way of leveraging existing type annotations for effective fuzzing. To the best of our knowledge, we are the first to test typed code for purposely-introduced inconsistencies.

9 Conclusion

This paper presents TypePatrol, the first adversarial testing approach that targets JavaScript and TypeScript libraries. The framework operates by instrumenting selected code locations within the libraries that includes security-relevant points. After this, we execute the developer-provided test suite, and collect the corresponding runtime behavior. Next, we mutate a subset of unit tests and execute the test suite again to simulate how an adversary might exercise the code and observe the effects i.e., whether any new behavior is triggered. The final step of TypePatrol is to compare the two sets of behaviors and to identify security-relevant type mismatches and analyze the effects.

We evaluate TypePatrol on 30 popular JavaScript and TypeScript libraries and find hundreds of type mismatches during adversarial testing. We find that with increased time budget, TypePatrol can uncover new security-relevant mismatches. At the time of writing, TypePatrol has resulted in ten pull requests in seven libraries. Developers have acknowledged our reports and merged nine of the pull requests demonstrating the practical applicability of TypePatrol.

References

- 1 Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5):66:1–66:36, 2017. doi:10.1145/3106739.
- 2 Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: Automatically testing javascript apis with asynchronous callbacks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1494–1505. ACM, 2022. doi:10.1145/3510003.3510106.
- 3 Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. Secbench.js: An executable security benchmark suite for server-side javascript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070, 2023. doi:10.1109/ICSE48619.2023.00096.
- 4 Carolin E. Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022. doi:10.1007/S10664-021-10094-2.
- 5 Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node.js vulnerabilities. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: <https://www.ndss-symposium.org/ndss-paper/nodemedic-fine-automatic-detection-and-exploit-synthesis-for-node-js-vulnerabilities/>.
- 6 Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, pages 1101–1127. IEEE, 2023. doi:10.1109/EUROSP57164.2023.00068.
- 7 Developer Community. Definitelytyped [online]. 2025. URL: <https://definitelytyped.org> [cited 2025-05-06].
- 8 Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 85–96. ACM, 2010. doi:10.1145/1831708.1831719.
- 9 Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *J. Syst. Softw.*, 157, 2019. doi:10.1016/J.JSS.2019.110398.
- 10 Yaniv David, Neophytos Christou, Andreas D Kellas, Vasileios P Kemerlis, and Junfeng Yang. Quack: Hindering deserialization attacks via static duck typing. In *the Network and Distributed System Security Symposium (NDSS)*, 2024.
- 11 DefinitelyTyped. <https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types>. Accessed: 2026-05-17.
- 12 Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 1–16, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660215.

- 13 Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 758–769. IEEE / ACM, 2017. doi:10.1109/ICSE.2017.75.
- 14 Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. Highly illogical, kirk: spotting type mismatches in the large despite broken contracts, unsound types, and too many linters. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):479–504, 2022. doi:10.1145/3563305.
- 15 Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1059–1076. IEEE, 2023. doi:10.1109/SP46215.2023.10179352.
- 16 Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL: <https://www.ndss-symposium.org/ndss-paper/auto-draft-207/>.
- 17 Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 991–1008. IEEE, 2025. doi:10.1109/SP61157.2025.00016.
- 18 Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, pages 121–134. USENIX Association, 2020. URL: <https://www.usenix.org/conference/raid2020/presentation/koishybayev>.
- 19 Erik Krogh Kristensen and Anders Möller. Reasonably-most-general clients for javascript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 83–93. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00026.
- 20 Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1193–1204. ACM, 2013. doi:10.1145/2508859.2516703.
- 21 Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 143–160. USENIX Association, 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/li-song>.
- 22 Libraries.io: JavaScript Packages Sorted by Dependent Count. https://libraries.io/search?languages=JavaScript&order=desc&platforms=npm&sort=dependents_count. Accessed: 2026-05-17.
- 23 Libraries.io: TypeScript Packages Sorted by Dependent Count. https://libraries.io/search?languages=TypeScript&order=desc&platforms=npm&sort=dependents_count. Accessed: 2026-05-17.
- 24 Xinrong Lin, Baojian Hua, Yang Wang, and Zhizhong Pan. Towards a large-scale empirical study of python static type annotations. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–425, Taipa, Macao, March 2023. IEEE. doi:10.1109/SANER56733.2023.00046.
- 25 Zhengyu Liu, Kecheng An, and Yinzhi Cao. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node.js template engines for malicious consequences. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4015–4033. IEEE, 2024. doi:10.1109/SP54263.2024.00121.

- 26 Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 304–315. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00045.
- 27 npm Registry. <https://www.npmjs.com>. Accessed: 2026-05-17.
- 28 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: neural type prediction with search-based validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 209–220. ACM, 2020. doi:10.1145/3368089.3409715.
- 29 Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324, 2015. doi:10.1109/ICSE.2015.51.
- 30 Michael Pradel and Koushik Sen. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 519–541. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.519.
- 31 Ingkarat Rak-Amnourykit, Daniel McCrean, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 57–70, 2020. doi:10.1145/3426422.3426981.
- 32 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015. doi:10.1145/2676726.2676971.
- 33 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 76–100. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.
- 34 Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. Test generation for higher-order functions in dynamic languages. *Proc. ACM Program. Lang.*, 2(OOPSLA):161:1–161:27, 2018. doi:10.1145/3276531.
- 35 Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *USENIX Security Symposium*, 2023.
- 36 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007. doi:10.1007/978-3-540-73589-2_2.
- 37 Dominic Troppmann, Aurore Fass, and Cristian-Alexandru Staicu. Typed and Confused: Studying the Unexpected Dangers of Gradual Typing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1858–1870, Sacramento CA USA, October 2024. ACM. doi:10.1145/3691620.3695549.
- 38 Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. Stubbifier: debloating dynamic server-side javascript applications. *Empir. Softw. Eng.*, 27(7):161, 2022. doi:10.1007/S10664-022-10195-6.
- 39 Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2951–2968. USENIX Association, August 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/xiao>.
- 40 Yishun Zeng, Yue Wu, and Chao Zhang. Typenfuzz: Dynamic type-aware object dependence graph guided fuzzing for javascript library bug discovery. *ACM Transactions on Software Engineering and Methodology*, 2025.

- 41 Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 995–1010. USENIX Association, 2019. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman>.