CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Detecting Prototype Pollution on The Web

Master's thesis in Computer science and engineering

Samuel Kajava

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Detecting Prototype Pollution on The Web

Samuel Kajava

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Detecting Prototype Pollution on The Web
Samuel Kajava

Supervisor: Andrei Sabelfeld, Department of Computer Science and Engineering
Examiner: Gerardo Schneider, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Detecting Prototype Pollution on The Web
SAMUEL KAJAVA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Prototype pollution is a vulnerability exploiting the inner workings of the JavaScript programming language. Being the most used language in the world, the security risks cover a large body of applications. JavaScript powers both web servers and client-facing websites, making it important to mitigate threats such as prototype pollution. Prior work has made significant strides on the server-side, creating a gap in user-facing applications.

The lack of research on client-side prototype pollution needs to be addressed, as vulnerable applications can be exposed to threats such as cookie stealing or cross-site scripting – all from visiting a malicious link. The goal with this thesis is to mitigate this threat by researching the prevalence of prototype pollution and defining a practical approach for securing web pages on the internet.

While progress has been made on the client-side, we have identified issues with existing approaches, such as limited code coverage, reliance on outdated software, and costly analysis. We resolve these issues by introducing a novel approach for detecting client-side prototype pollution. This is accomplished through complete source code retrieval, utilization of state of the art analysis tools, and readily available means to verify identified vulnerabilities, effectively eliminating any false positives. These steps complement each other to define a reliable approach for detecting prototype pollution vulnerabilities.

We realize our approach by creating a multi-stage framework for detecting prototype pollution in client-side JavaScript. Utilizing web crawling, we obtain all of the source code for a given website, ensuring full code coverage. To analyze and find vulnerable code on the website, we use CodeQL-powered static analysis for the first time in client-side prototype pollution research. The static analysis outputs a number of candidates which may lead to prototype pollution, which are then manually verified in a real-world setting by testing the vulnerability on the target website.

Our key contribution is our novel approach for detecting prototype pollution using our multi-stage approach. Our framework advances the state of the art by showing that static analysis is indeed feasible and more robust for client-side JavaScript compared to previous works. Our evaluation supports this claim by finding and verifying 28 domains vulnerable to prototype pollution. We demonstrate that our approach is viable by a performance evaluation and empirical comparison to the related work.

Keywords: JavaScript, security, prototype pollution, static analysis, CodeQL

Acknowledgements

I would like to extend my greatest appreciation for the support provided by my supervisor, Andrei Sabelfeld, whose insights, feedback, and open communication has helped tremendously. Furthermore, Benjamin Lundblad has provided great advice and has been great to discuss problems and ideas with. Last, but not least, SiKai Lu, Musard Balliu and Diogo Correia from KTH have all been great to collaborate with; exciting things ahead. Thank you.

Samuel Kajava, Gothenburg, 2025-01-09

Contents

List of Figures

List of Tables

1

Introduction

JavaScript is the world's most popular programming language and is responsible for powering the web [1]. This object-oriented language enables web developers to control contents, interactivity and styles for the websites they create in a highly flexible manner. This flexibility allows developers to create complex websites with capabilities such as interactive user interfaces, seamless interaction with various internet technologies, all within a comparatively small time-frame when compared to other languages. As a result, JavaScript has become the de-facto standard for web development over recent years. This reliance on JavaScript naturally induces great responsibility considering the scale of the web. For critical sectors such as finance or health care, JavaScript must go beyond its' core responsibility of ensuring functionality to serve as a core technology in mission-critical applications.

The same flexibility and power that make JavaScript imperative for web development also make it an attractive target for attackers – its ubiquitous deployment exposes it to a wide variety of security concerns. Vulnerabilities can arise from misconfigured systems, insecure coding practices or bugs in the use of third-party libraries. Since JavaScript often runs directly in the users' browsers, attackers can exploit such vulnerabilities to execute unauthorized code, manipulate website contents, or steal sensitive information. Moreover, attackers constantly find ways to improve their techniques, which makes it essential to identify and mitigate vulnerabilities proactively. This ongoing challenge introduces a need for robust security measures, including use of analysis tools and secure coding practices in order to stay vigilant.

1.1 Prototype pollution

An important emerging type of vulnerability is prototype pollution [2], which exploits the underlying mechanisms responsible for making JavaScript object-oriented. JavaScript accomplishes object-orientation through the *prototype chain* – which means that each object contains a link to its predecessor in order to access inherited features by querying said link (called the *prototype*). To understand what prototype pollution is, we begin with explaining prototypes.

Consider the example, based loosely on MDN Web Docs [3], in Listing 1; here a string is defined on line 1. The string contains a `__proto__`-field which accesses its prototype. The prototype includes useful builtin functions as shown on lines 7-8, but we can also observe that the string itself does not contain the function used in

the example, but is rather defined on the prototype. The example on line 7 utilizes the prototype chain, while line 8 explicitly calls the prototype. As discussed above, each object contains a link to its prototype in order to share functions and other properties. When we call the `includes(...)` function, the JavaScript engine first checks if the `str` variable has a reference called `includes`, finds out that it does not, and continues its traversal of the chain and finds the field in the prototype object. This is useful in the sense that every string does not need to define every function it has access to, but can rather use the shared prototype object. On line 11, the example adds a new function to the `String.prototype` – `toNumber`; an example function that lets the developer convert strings into numbers.

```
1  let str = "hello world";
2
3  // Primitive values have a prototype which is accessed via `__proto__`
4  str.__proto__ == String.prototype; // 'true'
5
6  // Prototypes provide useful builtin functions
7  str.includes("world"); // accessed via the prototype chain
8  String.prototype.includes.call(str, "world"); // verbose equivalent
9
10 // It is possible to add "builtin" functions to the prototype directly
11 String.prototype.toNumber = function () { return parseInt(this); };
12 let strNum = "  123 ";
13 strNum.toNumber() == 123; // 'true'
```

Listing 1: Demonstration of prototypes in JavaScript.

If we consider lines 11-13 in Listing 1, it is apparent that the added function `toNumber` exists on variables created *after* the line is executed. If an attacker is able to achieve such behavior, the prototype is said to be polluted. The example in Listing 1 is however not enough for achieving prototype pollution, since the example is hard coded. Another attribute of JavaScript is that the language is dynamic. This partly means that it is possible to change behavior of functions during code execution. One way of achieving this is called *dynamic property access*, which allows developers to access objects through bracket notation, i.e., `obj["prop"]`. Furthermore, the `"prop"` value can come from a variable, which in turn might be controlled by an attacker through e.g., `obj[userControlledInput]`. We consider the implications of this fact in Listing 2, where the URL field in the browser is used to add fields to the prototype.

Listing 2 contains a dummy database object on lines 2-9 which contains two users. There is also a function for adding or updating fields to the database on lines 12-17, which is accomplished through dynamic property access via bracket notation. As previously discussed, it is possible to update the prototype using the `__proto__` as property name. On line 23, the user's name, field to change, and value to add is extracted. Here, the user input provides `__proto__` as username when calling the `updateUser` functoin, and consequently pollutes the prototype by adding an `isAdmin` property to the prototype. This is confirmed on lines 29-30, where an object created after the function call resolves the potentially dangerous `isAdmin` field.

```
// Dummy database for demonstration purposes
let db = {
    "alice": {
        id: 0
    },
    "bob": {
        id: 1
    },
}

// Update or add a field to the user.
function updateUser(db, username, field, value) {
    if (db[username] == undefined) {
        db[username] = { id: newId() };
    }
    db[username][field] = value;
}

// API for accessing the URL
let url = window.location;
// parseUserInput extracts the username, field and value
// from `url.com/?username.field=value`
let [username, field, value] = parseUserInput(url);

// example:    __proto__, isAdmin, true
updateUser(db, username,  field,   value);

// Check for prototype pollution
let obj = {};
console.log(obj.isAdmin); // 'true'
```

Listing 2: Prototype pollution vulnerable code

Apart from the fictional example just described, the consequences of prototype pollution have shown to result in remote code execution [4], denial-of-service [2] and cross-site scripting [5] for example, further motivating the need for securing websites from this vulnerability.

1.2 Goal and research questions

Given the prominence of prototype pollution, we define our goal for this thesis as an effort to provide a practical approach for detecting prototype pollution in client-side JavaScript. We accomplish this by answering two research questions:

RQ1: *What constitutes a practical approach for detecting prototype pollution vulnerabilities in client-side JavaScript?*

This question aims to define an effective detection approach by adapting previous works and adding means for automatic verification. Addressing

this question will provide actionable insights for developers while laying the groundwork for future research by identifying areas for further improvement.

RQ2: *How prevalent are prototype pollution vulnerabilities in client-side JavaScript?*

This question evaluates the real-world impact of the approach by measuring the frequency of prototype pollution vulnerabilities across the web. The findings will shed light on the severity and scope of the problem, emphasizing its significance in the broader context of web security.

1.3 Challenges

Several prior contributions [2], [4]–[11] have studied prototype pollution. However, most of these works focus on server-side JavaScript (code running on servers as opposed to in the browser), leaving a gap in user-facing client-side code. The small number of contributions on client-side prototype pollution [5], [6] relies heavily on techniques yielding issues with code coverage which in turn result in a lack of robustness. These contributions have provided encouraging results, finding several zero-day vulnerabilities (ones which have not yet been found at the time of discovery). Both of these approaches rely heavily on dynamic analysis in the sense that they only consider what is on the loaded website at the time of analysis. This highlights the coverage issue, but also uncovers the reliance on modified tools; in order to correctly identify and exploit prototype pollution, modified browser instances are needed, which makes it less useful from a developer's point of view since they would need to apply a patch on their local browser.

1.4 Contributions

They key contribution we provide in this thesis is a multi-stage framework for detecting prototype pollution. For the first time in client-side prototype pollution research, we introduce static analysis with CodeQL as part of our approach. In this way, we overcome the difficulties with coverage and reliance on modified tools. Static analysis allows us to maintain all source code for a given website in scope to solve the coverage issue. This allows us to detect prototype pollution using a readily available tool whilst not relying on a modified browser. However, static analysis alone is not sufficient, as it is prone to produce many false positives [6]. We therefore apply a verification step to ensure any potentially vulnerable website is verified in a browser. Hence, our approach is practical – every stage is automatic, and it does not rely on any modifications to any browser. This yields a practical approach for detecting and verifying prototype pollution. We detected and identified 28 websites vulnerable to prototype pollution using our framework.

1.5 Overview

This thesis represents the approach for delivering answers to the above research questions. In order to do so, the text is structured as follows:

Chapter 2 provides an overview of the related technical details to prototype pollution and ways of analyzing systematically. Chapter 3 describes the approach of combining static analysis with dynamic testing to identify prototype pollution vulnerabilities. In Chapter 4, the approach is applied to answer the introduced research questions, followed up by a discussion of the results and possible mitigations in Chapter 5. In Chapter 6, related work is reviewed and compared to this thesis, before finally concluding the work in Chapter 7.

2

Background

This chapter covers the context for our research in this thesis. Given the introduction to prototype pollution described in Chapter 1, we begin with a motivating example of a real world prototype pollution vulnerability. After that, we cover relevant techniques used for detecting the vulnerability.

2.1 A motivating example

We begin with a motivating example of prototype pollution might be exploited. Consider the code in Listing 3.

```
1 let transport_url = config.transport_url || defaults.transport_url;
2 let script = document.createElement('script');
3 script.src = `${transport_url}/example.js`;
4 document.body.appendChild(script);
```

Listing 3: Example of prototype pollution, provided by PortSwigger Academy [12].

This example shows a snippet of library code which fetches an example script from another URL. The URL is dependent on the `config` object, and jumps to a default value in case it is not defined (line 1). However, the target website happens to be vulnerable to prototype pollution. The fallback `defaults.transport_url` can be avoided by injecting a value in the prototype's `transport_url`. If the prototype has the field, the prototype chain will resolve it on line 1 and therefore skip the fallback despite `config` not having it defined as the developer might have wanted.

```
1 // payload: url.com/__proto__[transport_url]=data:,alert(1);
2 script.src = `alert(1);///example.js`;
3 document.body.appendChild(script);
4 // becomes <script>alert(1);</script> on the website
```

Listing 4: The resulting code when Listing 3 is exploited.

As demonstrated in Listing 4, the attacker may enter

```
url.com/?__proto__[transport_url]=data:,alert(1);//
```

in the URL as their *payload* (the data required for exploiting the application). The example uses `alert(1)` as part of the payload, but it can contain arbitrary JavaScript,

which makes it dangerous. With this in mind, it is clear that the implications of prototype pollution are serious. Several prior contributions show that this indeed is the case, as they have all found real world examples of prototype pollution leading to exploits of this kind [4], [7], [10].

2.2 Detecting prototype pollution

Detecting vulnerabilities in general is possible through various methods. Employing techniques such as analysis of source code before and during runtime are common.

2.2.1 Static analysis

Analysing software by examining the source code in text form, i.e., statically, is the practice of static analysis [13]. By observing code without compiling or otherwise running it, makes it possible to spot bugs or vulnerabilities efficiently. In its simplest form, static analysis is so-called manual auditing, where a human looks at the source code of a project to spot patterns which may introduce vulnerabilities, but as the authors point out, it is a time-consuming task. Moreover, this approach requires in-depth knowledge of security risks and will wary between any two people, making it inconsistent at best. To remedy this, automated tools exist, operating similarly to a compiler in the sense that given an input, the tool outputs any errors (vulnerabilities) that occurred during analysis.

Static analysis is a convenient technique in the sense that it does not depend on any runtime, just the source code [13]. However, considering the dynamic nature of JS, a purely static approach introduces challenges since the runtime itself introduces a complex environment [14]. Apart from that, static analysis is prone to providing false positives, as it has no means for verifying what it has detected [5], [6].

2.2.2 Taint analysis

Considering the issue where pure static analysis is unable to reason about malicious user-controlled input can be handled by taint analysis [15]. This approach reasons about information flows, e.g., how a tainted value (user-controlled) propagates throughout the execution of a program. By tracking the path a value takes in an environment, it is possible to inspect which parts of the source code is possibly affected by the tainted, user-controlled, value. More concretely, the process uses the concepts of *taint sources* and *taint sinks* to perform analysis. A taint source is data which is initially labeled as tainted, this can be input from I/O for instance – user-controlled. The taint sink is also data which deems an application vulnerable in the case where a tainted value reaches it.

Consider the example in Listing 5. Here, a merge function is defined; it copies all properties from the source object to the target. Lines 1 and 2 indicate a taint source. Since any bracket notated property access can modify the prototype, it is therefore a possible taint sink, as labeled on line 8.

```
1  const source = fetchUserData(); // #taint
2  // assume source = { __proto__: { toString: 0 }}
3  const merged = merge({}, source);
4
5  function merge(target, src) {
6    for (let i in src) {
7      target[i] = src[i];
8   // --------- #sink
9    }
10   return target;
11 }
```

Listing 5: Vulnerable merge function annotated with labels used for taint tracking.

Step	Line	Expression	Comment
1	1	`const source = fetchUserData();`	Source `source #taint`
2	3	`const merged = merge({}, source);`	Taint step `merged #taint`
3	6	`for (let i in src)`	`i = '__proto__'`
4	7	`target['__proto__']={'toString':0}`	`Object.prototype #sink`

Table 2.1: Execution flow for vulnerable code indicated by a tainted value from a taint source reaching a taint sink.

To verify this tainted flow, the execution of the snippet is displayed in Table 2.1. The code is indeed vulnerable, since the tainted value from the source (`fetchUserData()`) propagates to the taint sink (`Object.prototype`) via prototype pollution.

The example just discussed provides an intuitive explanation of how taint analysis works, and is not based on any technology in particular. There are tools for implementing such techniques.

2.2.3 Taint analysis with CodeQL

One option for performing static taint analysis is CodeQL, a language and toolchain for analysing source code [16], [17]. It utilizes *variant analysis* for describing vulnerabilities which can be used for identifying similar problems in a codebase. The way of describing vulnerabilities is done using QL, a query language which allows users to write queries that can be run on the project at hand. More specifically, CodeQL treats source code as data, and builds a relational database for the codebase, which can be queried to find vulnerabilities. As introduced, it also supports data-flow analysis for reasoning about tainted flows.

CodeQL uses a data flow graph for performing information-flow analysis, which is a representation of how data travels throughout the runtime execution of a program [17]. This avoids the dynamic part discussed earlier, making it suitable for performing analysis of JS when a runtime is not provided, for example. With this representation in place, queries can use it to query the CodeQL database in order to look for vulnerabilities where tainted sources reach tainted sinks.

Circling back to the vulnerable code in Listing 5, CodeQL can be used to find the vulnerability, addressing the limitations of pure static analysis in Section 2.2.1, while avoiding the dependency of a runtime, as presented in Section 2.2.2. Consider the query described in Listing 6. Here, a query is built to address the problems with bracket notated property access prone to prototype pollution.

```
1  import javascript
2  import DataFlow::PathGraph
3  import PrototypePollutingCommon
4
5  from PrototypePolluting::Configuration cfg,
6    DataFlow::PathNode source,
7    DataFlow::PathNode sink
8  where cfg.hasFlowPath(source, sink)
9  select sink, source, sink,
10   "This assignment may alter Object.prototype if a \
11   malicious '__proto__' string is injected from $@.",
12   source.getNode(), "here"
```

Listing 6: CodeQL query for detecting prototype pollution, adapted from a previous contribution; Silent Spring [18].

While the query lacks some of the code required for it to run, it is still possible to reason about its functionality. The syntax of a CodeQL query looks like `from /* Vars */ where /* Logic */ select /* Expressions */` [16].
The case for the query in Listing 6, the query can intuitively be described as "*based on a source and a sink, is there a flow path between them such that it is vulnerable to prototype pollution?*". The configuration defined on line 5 is what is used for determining whether something is possibly vulnerable to prototype pollution. One possibility is where a tainted source reaches a sink which accesses a property using bracket notation.

Static analysis with CodeQL has seen use in prior contributions [4] on server-side prototype pollution, but has not yet been applied to client-side focused research.

3

A Hybrid Approach for Detecting Prototype Pollution

Detecting prototype pollution requires a number of processing steps before obtaining a result. This chapter describes the hybrid approach used for detecting prototype pollution, realized as an end-to-end pipeline. The term "hybrid" here refers to the integration of static and dynamic analysis methods, which collectively enable detection, candidate selection, and verification. The following overview provides a description of the pipeline's stages and their respective contributions to the detection process.

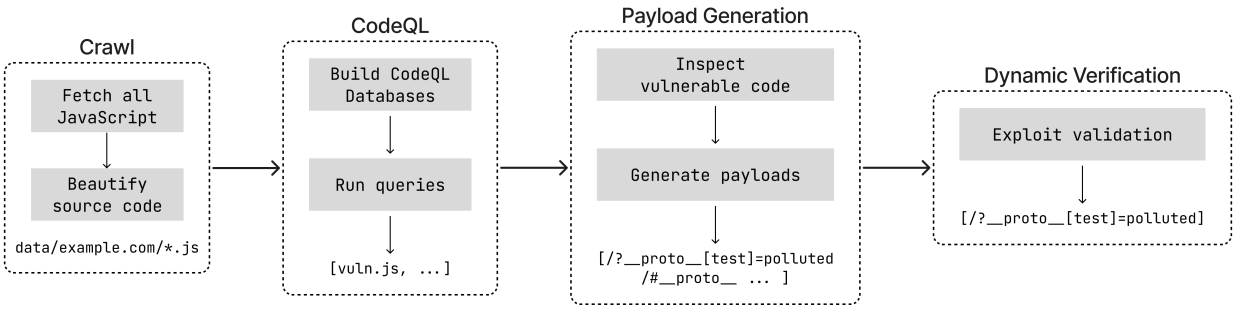

Figure 1: Framework design. Going from left to right, the gray boxes indicate the high-level workflows, the text at the bottom of each step shows the output.

An overview of the approach is described in Figure 1, showcasing the process from going from a list of URLs to a list of vulnerabilities. The first step – crawling – saves all domains into separate directories. The CodeQL step builds databases for each domain, runs queries on them, and saves the results in a file with information regarding which files are potentially vulnerable. The next step analyses the output from CodeQL and generates payloads, before finally dynamically verifying the payloads in order to eliminate false positives. The final output is a list of vulnerable domains with their respective payloads. The upcoming sections describe the intricacies of each step.

3.1 Challenges

As we will see in this chapter, detecting prototype pollution is not a trivial task. Many websites try to mitigate the threat of prototype pollution by using firewalls. This is accomplished by blocking HTTP requests containing the typical strings like `__proto__`. While this is effective, the prototype pollution still remains on the website. To remedy this, we employ techniques for avoiding the firewall using builtin tools provided by web browser (see Section 3.5).

Some websites also tend to add checks where the user is prompted to confirm that they are human (captchas). This problem is hard to solve automatically and therefore leave some webpages unavailable for analysis.

3.2 Pre-processing

Domain retrieval. The input is a list of domains up for analysis. Each domain is downloaded separately, retrieving all of the source code that a regular browser would. This is accomplished by observing the incoming network traffic, where each response containing JavaScript or HTML is saved to disk, grouped by domain, e.g., `chalmers.se` and `kth.se` would be contained in separate directories.

Ethical considerations. It is important not to overwhelm the target domain. The crawler visits each website ones, and therefore mimics the standard use case of a regular user visiting the page.

De-minification. Detecting prototype pollution statically requires pre-processing. Source code on the web is often *minified*, meaning that it is compressed as much as possible in order to reduce the size of the outgoing network traffic. While retaining functionality, the code is less optimal for static analysis, as the whole program is defined on one line. To remedy this, every domain is *beautified* to reacquire indentation and line breaks one would typically see in source code. An example of the output of a *minifier* is shown in Listing 7.

```
< function fib(n) {
<     if (n < 2) {
<         return n;
<     }
<     return fib(n-1) + fib(n-2);
< }
---
> function fib(f){return f<2?f:fib(f-1)+fib(f-2)}
```

Listing 7: Diff of the original and minified versions of a recursive Fibonacci function.

This step is important for the next, *database creation*, since CodeQL by default ignores minified files. Furthermore, it helps the user understand the results provided by the analysis in the end since the messages about vulnerable code will be linked to line in the source code, rather than a column range on a long line.

Database creation. This approach utilizes CodeQL queries for detecting prototype pollution. The mechanics behind this is described in the upcoming step, but in order to run queries, the domains need to be represented as a database (recall Section 2.2.2). This step simply builds a relational CodeQL-database per domain. It is important to keep the separation between domains, as the database would otherwise possibly form relations between references across projects.

3.3 Candidate selection with CodeQL

The detection process starts by identifying candidate flows that may lead to prototype pollution. These flows are flagged using CodeQL queries designed specifically for prototype pollution. Building on the foundation of *Silent Spring* [4], the queries are adapted to focus on client-side sources. This adaptation extends the scope beyond the server-side focused contribution, providing insights into browser-specific functionality.

1. Taint Flow Analysis. Taint flow analysis is essential for candidate selection. By tracking data flow from sources (attacker-controlled data) to sinks (vulnerable points prone to modifying the prototype), CodeQL is responsible for flagging flows matching certain criteria:

- **Sources.** Input points which may be influenced by an adversary:
  - Query parameters, i.e., the parts after the `?` token in a URL such as `domain.com/?query`: `window.location.search`, `URLSearchParams` etc.
  - Fragments, parts preceding the `#` token, i.e., `domain.com/#fragment`. These are accessed by using the `window.location.hash` property.
  - General URL parsing, APIs which process the whole URL, `URL.parse()`, for instance.
- **Sinks.** Expressions that write to properties enabling prototype pollution:
  - Direct access to the prototype, e.g., `Object.prototype`.
  - Dynamic property access, e.g., `obj[k] = val` where `obj`, `k` and `val` are adversary controlled.

2. Query Mechanism. CodeQL queries, much like SQL, defines a set of rules to select entries from a database. In this approach, the queries are used to detect tainted flows which are potentially vulnerable to prototype pollution. It does so by using the defined sources and sinks described above to detect if there are any paths between any. In the case where there is a source reaching a sink, the query returns a result, i.e., a candidate flow. Below, in Listing 8, is a simplified example of what a query may look like.

3. Output. The output of this stage is a list of candidate flows. Each flow contains a sequence of steps describing the data path from source to sink, where each step contains information about where it is located on disk and what line is affected. Additionally, metadata regarding what type of source is affected is also supplied.

```
from Config cfg, Source src, Sink sink
where cfg.hasFlowPath(src, sink)
select src, sink, "Flow indicates possible prototype pollution"
```

Listing 8: Pseudo-code representation of a CodeQL query.

4. Challenges in Candidate Selection. While the selection process proves suitable as an initial step in detecting prototype pollution, it is not sufficient on its own, for several reasons:

- **Dead code.** Since CodeQL is, by design, static, it is difficult to know if the vulnerable source code will be executed without a runtime. This is the case for so-called *dead code* – code which exists but is never reached during program execution.
- **Minified and/or obfuscated context.** Minified code can obscure relationships between sources and sinks, making the previous formatting steps important for creating an optimal environment for analysis.
- **False positives.** There are legitimate use cases for e.g., dynamic property access, provided proper sanitization. It is hard to generalize detection of sanitization, and therefore the static analysis may improperly flag safe code guarded by input validation, for instance.

Given these challenges, the candidates need to be verified dynamically in order to eliminate false positives, which subsequent stages of the pipeline aim to accomplish.

3.4 Payload generation

The payload generation stage builds on the candidate flows identified during static analysis. Each flow is transformed into one or more payloads to test the possibility of prototype pollution. All payloads are formed with the threat model in mind – the attacker controls the URL. The produced payloads are structured as `candidate.org/payload`, where `payload` represents a URL query or fragment designed to exploit the dynamic property assignment (i.e., through bracket notation) in JS.

Payload creation by pattern matching. Payload generation relies on pattern matching algorithms that map the candidate flow to common prototype pollution patterns. This approach is adopted on the basis of previous contributions from both research and community efforts:

- **Arteau's patterns [2].** Demonstrates prototype pollution through dynamic property access.
- **BlackFan's collection of payloads [19].** Real-world examples of payloads proven to be vulnerable.
- **ProbeTheProto [5].** The first contribution to prototype pollution in client-side JavaScript, with many results containing various payloads.

Each candidate flow is analyzed line by line, extracting common APIs prone to prototype pollution. The payload generator then applies templates to construct URLs that may exploit the candidate flow. Examples include:

- From `window.location.search`:
  - Candidate: `obj[location.search.split("=")[0]] = value;`
  - Payloads: `[?__proto__[key]=val, ?constructor[prototype][key]=val]`
- From `window.location.hash`:
  - Candidate: `let k = location.hash.substring(1); obj[k] = value;`
  - Payloads: `[#__proto__[key]=val, #constructor[prototype][key]=val]`

By utilizing insights from previous works, this stage ensures that the payloads are based on ground truth, while extending the field by applying the preceding static analysis on client-side JavaScript.

Preparing for verification. The use of static analysis alone is insufficient, as discussed above. Therefore, the generated payloads from this step serve as input for the upcoming: dynamic testing.

3.5 Dynamic testing

The final stage of the pipeline takes the candidate payloads from the previous step as input and automatically visits each URL to validate whether prototype pollution has been achieved. This acts as a final filter for eliminating false positives from previous steps, as it only flags domains as vulnerable if the proposed payload indeed causes prototype pollution.

Strategy. For each URL payload, the dynamic step spawns a Chrome browser with access to the DevTools API provided by the browser. When the website is fully loaded, it is assumed that the payload has been handled by the application. As we discussed earlier in Section 3.1, some websites block requests containing payloads. In order to increase the amount of domains available for analysis, we do not directly include the payloads in the URL, but rather add them before loading the page once the source code has been fetched.

The next step involves using the provided DevTools API to open a console on the webpage, create a new object, and inspect its properties. If the field provided by the payload is defined and corresponds to the expected value, the website is flagged as vulnerable to prototype pollution.

This strategy can be manually verified as well; Figure 2 illustrates what the program observes. By observing the payload in the URL field, it is apparent that the payload is reflected in the prototype. This is considered a true positive, since the field in the payload is reflected in new objects.

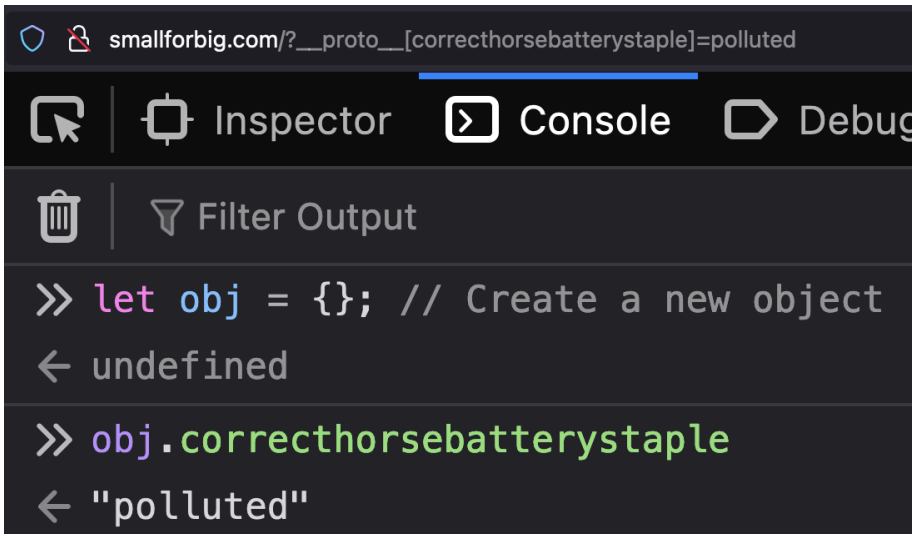

Figure 2: Manual equivalent of dynamically verifying prototype pollution.

Ethical considerations. Each found vulnerability will be reported to the owner of the affected websites, ensuring responsible disclosure. We also ensure that each exploit we generate is benign by only polluting test values without any side effects. This ensures that the website is not affected by the exploit, and just exists as a proof-of-concept.

4

Evaluation

This chapter covers the evaluation of the proposed approach in Chapter 3 in order to answer the research questions introduced in Chapter 1. Using the experimental setup in Section 4.1, the approach is analyzed based on the evaluation criteria defined in Section 4.2, before discussing the implications and inherent answers to the research questions in Section 4.3.

For convenience, the research questions are repeated here:

RQ1: *What constitutes a practical approach for detecting prototype pollution vulnerabilities in client-side JavaScript?*

RQ2: *How prevalent are prototype pollution vulnerabilities in client-side JavaScript?*

4.1 Experimental setup

This section describes the chosen baseline and datasets for the evaluation of the presented approach in Chapter 3. All experiments were conducted on Ubuntu 24.04 LTS with an Intel i9 13900 processor and 32 GB of memory.

4.1.1 Baseline

As a baseline for evaluating false negative rates, the previous client-side oriented contribution, ProbeTheProto [5] is used. The publicly provided results are used as ground truth for investigating the accuracy of the approach presented in this thesis.

ProbeTheProto is a dynamic analysis tool for not only detecting prototype pollution, but adds an additional step of finding gadgets, i.e., otherwise benign code which is exploited using prototype pollution. Despite this, the dataset will remain relevant since it still acts as a set of true positives.

4.1.2 Datasets

The following datasets are used when evaluating this contribution:

- **ProbeTheProto.** This contains 369 vulnerable domains after filtering out patched or unresponsive domains from the original results of 2,738 vulnerable websites. It is used for comparing our approach to ProbeTheProto to determine false negative rates.

- **10K websites.** This contains ten thousand unique domains retrieved from the Tranco List [20]. These domains are used for determining the prevalence of prototype pollution on the web. Furthermore, the rates acquired from analysing this data is used for comparing to previous client-side contributions in order to determine whether the proposed approach is practical.

4.2 Evaluation criteria

This section defines the evaluation criteria used to assess the proposed approach for detecting prototype pollution in client-side JavaScript. The criteria are divided into three main categories: *Baseline comparison*, *Prevalence analysis*, and *Performance overhead*.

4.2.1 Baseline comparison

The baseline comparison measures how well the proposed approach detects known prototype pollution vulnerabilities, using ProbeTheProto as ground truth. This comparison is mainly used to answer **RQ1**, to determine whether static analysis with dynamic verification is a reasonable approach for detecting prototype pollution.

Metrics

1. **False negatives (FN).** Domains which our tool falsely marks as not vulnerable to prototype pollution.
2. **True positives (TP).** Domains which our tool successfully marks as vulnerable.
3. **Recall.** The fraction of true positives that were obtained from the ground truth data: $\frac{TP}{TP+FN}$.

4.2.2 Prevalence analysis

The prevalence analysis measures how frequently prototype pollution occurs on the web, using the 10K websites dataset.

Metrics

1. **Discovered vulnerabilities (DV).** The number of confirmed new vulnerabilities detected.
2. **Detection Rate (DR).** The rate of confirmed vulnerabilities detected in the wild.
3. **Comparison with prior work.** Comparisons of DR with ProbeTheProto.

The metrics defined above are useful for both research questions, since providing a detection rate on par with previous work will show whether the approach is practical (**RQ1**), and the acquired prevalence will directly answer **RQ2**.

4.2.3 Performance analysis

The performance analysis measures how long, on average, it takes to analyze a domain. The dataset used is the discovered domains vulnerable to prototype pollution combined with the same amount of domains which are not vulnerable.

4.3 Results

This section covers the results from running the experiments and discusses them based on the provided research questions.

4.3.1 Baseline comparison

The baseline comparison with ProbeTheProto reports 38 false negatives out of 369 domains after filtering out patched and unresponsive websites (originally 2738). Out of the false negatives, 6 of them failed due to faulty JavaScript, resulting in the CodeQL step in the pipeline failing. The results are shown in Table 4.1, and yields a recall of $\frac{331}{369} = 0.89$.

Table 4.1: Recall from the baseline comparison

Metric	**Value**
FN	38
TP	369
Recall	0.89

The baseline results highlights two important aspects of our approach which both relate to **RQ1**; namely:

1. **CodeQL in client-side JavaScript.** Erroneous JavaScript results in a failure in creating a CodeQL database, which leads to the affected domain being ignored in the later steps of the pipeline. This lack of robustness affects the coverage of the tool, as such domains would be ignored. The baseline comparison highlights this, albeit small, caveat, since such domains would otherwise go unnoticed in the Wild. Furthermore, it underscores the importance of handling errors gracefully in client-side JavaScript, something that affects the practicality of the given approach.

2. **Areas of improvement.** The recall of 0.89 shows that this approach is able to detect prototype pollution despite the different strategies when compared to ProbeTheProto. Using CodeQL has its challenges in regards to syntax errors, but missed vulnerabilities can be analyzed to implement new patterns to the payload generation step, thanks to its modular design. This ease of adding new prototype pollution patterns allows for iterative additions, which benefits the practicality of this approach.

4.3.2 Prototype pollution prevalence

The results from running our pipeline on 10,681 unique domains are shown in Table 4.2. Some failed to analyze due to a number of reasons: (a) the JavaScript contained syntax errors, which CodeQL failed to analyze, (b) the website contained captchas and subsequently hindered us from running dynamic verification, or (c) our requests were blocked by a firewall. Out of the successfully analyzed domains, 28 of them were dynamically verified as vulnerable to prototype pollution.

Table 4.2: Summary of analysis results of the pipeline

Metric	**Value**
Total domains	10,681
Failed to analyze	356
Successfully analyzed	10,325
Discovered vulnerabilities (DV)	28

As for the detection rate, compared to ProbeTheProto, despite the smaller dataset, the detection rates are roughly the same, as shown in Table 4.3.

Table 4.3: Metrics compared to ProbeTheProto [5]

Contribution	**#**	**DV**	**DR**
Our approach	10,325	28	0,2711%
ProbeTheProto [5]	1M	2738	0,2738%

To provide further insight into the prevalence of prototype pollution, we categorize the types of sources responsible for the prototype pollution in Table 4.4. String-based vulnerabilities come in the form of `__proto__['some string']`, which opens up for more critical consequences when compared to the numeric payloads, which only allows for numeric indexing, e.g., `__proto__[0]`. The reason for this relates to the possibility of overwriting undefined property accesses in the vulnerable application. If, for instance, a website tries to access a property, `obj.foo`, but returns `undefined`, it is possible to inject a user controlled value to that access (as illustrated in Chapter 2). The implications of such vulnerabilities are studied in ProbeTheProto [5] and Silent Spring [4] among others, and may lead to remote code execution (RCE), cross-site scripting (XSS), or denial-of-service (DoS).

Table 4.4: URL-based source types

Source	**string-based**	**numeric**	**#**
`Query`	11	16	27
`Fragment`	0	1	1

RQ2 - Prevalence. The similar DR values observed between our results and those from ProbeTheProto, along with implications reported in prior research, indicate that prototype pollution is prevalent in client-side JavaScript, leading to consequences such as XSS, RCE, and DoS. Thus, our findings reinforce those of ProbeTheProto, confirming the widespread nature of prototype pollution.

4.3.3 Performance analysis

Given the dataset of 28 vulnerable domains and 28 non-vulnerable domains, it took 32 minutes, yielding an average of 0.7 minutes per domain, as shown in Table 4.5. The crawling step is omitted as it is too dependent on internet speed.

Table 4.5: Breakdown of analysis time.

Step	**Time**
Beautification	8 min
CodeQL	8 min
Payload generation and verification	15 min
Total time	40 min
Avg. time per domain	0.7 min

Considering the time spent per domain, we deem the approach practical from a performance viewpoint, especially since it is fully automated. Moreover, since each domain is isolated from the rest of the dataset, it is possible to increase the parallelism if faster analysis is required.

4.4 Evaluation takeaways

The evaluation of the proposed approach provides several insights into its effectiveness and areas for improvement as well as broader implications about prototype pollution in client-side JavaScript. This section covers the key takeaways, structured around the research questions.

4.4.1 RQ1 - Practicality of the approach

As we have pointed out previously in Section 2, static analysis is prone to high false positive rates. Our dynamic verification step addresses this by filtering out candidates which do not result in prototype pollution. Given that our approach yielded detection rate of 0.2711%, we show that static analysis with dynamic verification is indeed practical for detecting prototype pollution when compared to ProbeTheProto whom reached similar rate of 0.2738%. We, like ProbeTheProto, encountered challenges during analysis. CodeQL is unable to handle erroneous JavaScript, which may introduce missed vulnerabilities. ProbeTheProto encounters the same issue – considering its difficulties with coverage due to their dynamic detection approach.

4.4.2 RQ2 - Prevalence of prototype pollution

Our approach shows results on par with ProbeTheProto, with 28 vulnerabilities across 10,325 domains. This consistency confirms the prevalence of prototype pollution in client-side JavaScript. Furthermore, a significant portion of the identified vulnerabilities showcase string-based payloads. This indicates higher risks by enabling the manipulation of undefined property accesses.

The majority of results reveal that query-based payloads are more common. This insight suggests that URL queries are prone to prototype pollution and highlights areas for focused mitigation efforts.

One common mitigation technique is the use of firewalls and captchas proves effective for preventing attacks. On the other hand, the consequence of this results in a failure of analysis. This is an issue for any crawling-based approach, making it hard to avoid when doing large scale analysis. It is therefore not feasible to address within the scope of this thesis.

5

Mitigations

As shown by previous studies and this thesis, prototype pollution in client-side JavaScript is prevalent on the web. However, like most vulnerabilities, it is preventable, and several mitigation techniques have been proposed. Most of the proposed changes restrict the functionality of the affected code, and therefore need careful consideration before implementing them. In this chapter, we provide an overview of such mitigations and their consequences on the application at hand. Additionally, some examples include vulnerabilities from the evaluation results with a fix using mitigations.

5.1 Freezing the prototype

Albeit quite a drastic measure, it is possible to completely "freeze" the prototype from being altered altogether, and is introduced as a mitigation by Arteau [2]. By calling `Object.freeze(Object.prototype)`, the prototype is accessible, but future changes to it is impossible, which means that no properties or functions can be added [21]. In the case where third-party code is present, it is possible that those libraries depend on altering the root prototype, and would consequently cease to function properly, and is therefore not suitable in larger projects.

Another case where this mitigation would be unfeasible is for polyfills. A polyfill is specific JavaScript code responsible for providing functionality which is not natively supported in some browsers [22]. These often rely on modifying the root prototype, and would therefore break in cases where the prototype is frozen.

5.2 Using the `Map` object

Suggested by Arteau [2], dynamically accessing properties using bracket notation is commonly used for key-value storage, and is a common bug in application vulnerable to prototype pollution. If, instead, a developer wishes to implement such behavior, they may mitigate the threat of prototype pollution by explicitly initializing new objects as `Map` objects. If a malicious `__proto__` were to be injected, it would simply exist as a key-value pair in the given object. In most cases, this adaptation would eliminate the risk without loss of functionality, since implementing inheritance is typically not done in this way. It should be mentioned that this mitigation is not a

global fix, as each new object would have to be refactored to adhere to the mitigation, which can be cumbersome in larger projects.

```
/* --- Vulnerable --- */
const data = {};
const userInput = "__proto__"; // Attacker-controlled input

// Attacker modifies the prototype chain
data[userInput]["isAdmin"] = true;
delete Object.prototpe.isAdmin;

const another = {};
console.log(another.isAdmin); // `true`

/* --- Safe --- */
const data = new Map();
const userInput = "__proto__"; // Attacker-controlled input
const value = { isAdmin: true };

// Attacker tries to modify the prototype chain
data.set(userInput, value);

const another = {};
console.log(data.get(userInput).isAdmin); // `true`
console.log(another.isAdmin); // `undefined`, no prototype pollution
```

Listing 9: Using a `Map` for key-value logic removes the threat of prototype pollution.

Listing 9 illustrates how a refactor might look for implementing this mitigation. The first vulnerable part shows a problematic implementation for setting a user to admin, whilst the safe option only affects the `data` variable.

5.3 Null initialization

Another mitigation technique which disables prototype pollution for new objects is using `let obj = Object.create(null)` instead of `{}` [2]. This eliminates the prototype altogether from the object, and can therefore not be polluted. However, considering that the prototype chain is no longer available, the object loses functionality one might expect. Therefore, this mitigation technique must be carefully considered before using it – which otherwise may end up introducing bugs if the developer is unaware of the implications.

5.4 Changing JavaScript

Another mitigation is introduced as an ECMAScript proposal, aimed at avoiding access to prototypes through properties [23]. By implementing an opt-in feature for encapsulating prototypes, it is possible to delete the vulnerable properties. This

would remove the threat of prototype pollution, but needs a new way of accessing the properties to avoid breaking functionality. The proposed change for ensuring backwards-compatibility is adding APIs for accessing the prototype. To ensure that older applications remain functional, the proposal includes an additional step when parsing JS which automatically refactors existing code to respect the new restraint. If added to the ECMAScript specification, prototype pollution via bracket notation access would be impossible once implemented in the browser.

5.5 Firewalls

As shown in Chapter 4, firewalls are able to block networks requests containing fields prone to prototype pollution, this is also a viable option. If, for example, we enter `example.org/?__proto__[polluted] = true`, the request could simply be blocked. This hinders an attacker from polluting the prototype, and circumvents the issue. However, as we also discuss, this does not fix the issue at hand. Optimally, another mitigation technique should be used in combination with this one to increase safety.

6

Related work

In this section, we cover related works on prototype pollution, and highlight the differences between each contribution with ours.

6.1 A first look into prototype pollution

Olivier Arteau's paper *Prototype pollution attack in NodeJS application* [2], is widely regarded as the seminal paper on the topic of prototype pollution. It introduces the concept to the security research community by providing a technical overview of the vulnerability. Arteau systematically explains how prototype pollution occurs, how it can be exploited, and outlined potential mitigation strategies.

While useful as a reference for prototype pollution as a vulnerability, it lacks a systematic approach for detecting it. The difference between our approach and Arteau's is that our contribution provide a practical way of automatically detecting prototype pollution.

6.2 Silent Spring

Presented by Shcherbakov et al., the paper *Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js* [4] presents Silent Spring, a multi-staged framework for detecting and exploiting prototype pollution in server-side JavaScript.

Silent Spring introduces static analysis with CodeQL for server-side JavaScript, showing that it is useful for prototype pollution research. Our contributions introduce the ability to statically analyze client-side JavaScript using CodeQL by doing it for the first time. We accomplish this by modifying the Silent Spring queries. Furthermore, their approach is semi-automated, which introduces issues with studies on a larger scale. Our approach does not suffer from this drawback.

6.3 ProbeTheProto

Kang et al. presents ProbeTheProto in *Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites* [5], the first

client-side contribution in the prototype pollution research body. Using dynamic taint analysis, they study prototype pollution prevalence in one million websites.

ProbeTheProto studies client-side JavaScript, making it relevant for our research. Their approach differs from ours significantly due to their choice of using dynamic analysis. Dynamic analysis introduces problems with code coverage [5], while observing less false positives. We utilize static analysis combined with dynamic verification to achieve both high code coverage and elimination of false positives.

6.4 Follow My Flow

Follow My Flow: Unveiling Client-Side Prototype Pollution Gadgets from One Million Real-World Websites by Kang et al. [6] is a recent contribution for detecting and exploiting prototype pollution in client-side JavaScript. They conduct a novel approach for finding gadgets by taking defined values from non-vulnerable websites and applying them to vulnerable ones.

Follow My Flow utilizes a dynamic approach, which suffers from coverage issues, just like ProbeTheProto. Yet again, our reliance on static analysis and ensures we do not come across this caveat.

6.5 Dynamic fuzzing

Detecting prototype pollution for node.js: Vulnerability review and new fuzzing inputs by Zhou et al. [7] addresses weaknesses of fuzzing and static analysis approaches by improving prior fuzzing techniques.

By improving fuzzing techniques, the authors detects more vulnerabilities. Yet, relying solely on fuzzing can lead to false negatives [7]. Our approach avoids using fuzzing altogether.

7

Conclusion

This thesis proposes a multi-stage framework for detecting prototype pollution in client-side JavaScript. By using static taint analysis with CodeQL, for the first time in client-side JavaScript, we address issues with prior works in terms of code coverage, costly analysis, and reliance on modified browsers. Furthermore, we leverage dynamic verification of the candidates produced by the static analysis in order to remedy the issue of false positives.

Our approach discovered 28 sites vulnerable to prototype pollution from 10K domains, highlighting the prevalence of the vulnerability. Moreover, based on baseline comparisons with ProbeTheProto [5], we find that our approach is viable. Hence, we conclude that the approach we present is indeed practical for detecting prototype pollution in addition to providing insight to its prevalence on the web.

7.1 Ethical considerations and disclosure

Considering that the approach described in Chapter 3 is applied to live websites, it is vital to ensure no damage is done during analysis. Therefore, we carefully consider the impacts of each step in our approach. The crawling step visits each website once, and is therefore benign, as it replicates typical use and does not overwhelm the server hosting the website. The CodeQL step is ran completely locally, and has no affect on the website itself. Same goes for the payload generation. The dynamic verification step is responsible for verifying the exploits, and to ensure no damage is inflicted, the payloads are benign in the sense that they do not affect the websites. This is accomplished by the use of dummy values that are never accessed on the site except for verifying that the prototype has been polluted.

As for the found vulnerabilities, all are in the process of being reported to the respective site owner.

Bibliography

[1] Stack Overflow, *Stack Overflow Developer Survey 2024: Most Popular Technologies - Programming Languages*, Accessed: 2024-12-11, 2024. [Online]. Available: `https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language`.

[2] O. Arteau, "Prototype pollution attack in nodejs application," *North-Sec*, 2018.

[3] *Inheritance and the prototype chain: Mdn.* [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`.

[4] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node. js," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5521–5538.

[5] Z. Kang, S. Li, and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites.," in *NDSS*, 2022.

[6] Z. Kang, M. Lyu, Z. Liu, *et al.*, "Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites," in *2025 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2024, pp. 16–16.

[7] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[8] M. Shcherbakov, P. Moosbrugger, and M. Balliu, "Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 1800–1811.

[9] Z. Liu, K. An, and Y. Cao, "Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences," in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2024, pp. 121–121.

[10] H. Y. Kim, J. H. Kim, H. K. Oh, *et al.*, "Dapp: Automatic detection and analysis of prototype pollution vulnerability in node. js modules," *International Journal of Information Security*, vol. 21, no. 1, pp. 1–23, 2022.

[11] E. Cornelissen, M. Shcherbakov, and M. Balliu, "{Ghunter}: Universal prototype pollution gadgets in {javascript} runtimes," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3693–3710.

[12] *What is prototpe pollution?* Accessed: 2025-01-5, 2024. [Online]. Available: https://portswigger.net/web-security/prototype-pollution.

[13] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[14] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 499–509.

[15] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

[16] GitHub CodeQL Team, *About CodeQL*, Accessed: 2024-11-21, 2024. [Online]. Available: https://codeql.github.com/docs/codeql-overview/about-codeql/.

[17] GitHub CodeQL Team, *About Data Flow Analysis in CodeQL*, Accessed: 2024-11-21, 2024. [Online]. Available: https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/.

[18] KTH Language-Based Security Group, *PrototypePolluting.ql*, Accessed: 2024-11-21, 2024. [Online]. Available: https://github.com/KTH-LangSec/silent-spring/blob/a48fcff103db2610555f31b5311e2bd74c5adb40/codeql/js-queries/PrototypePolluting.ql.

[19] BlackFan, *Client-Side Prototype Pollution*, Accessed: 2024-11-14, 2024. [Online]. Available: https://github.com/BlackFan/client-side-prototype-pollution/tree/master/.

[20] Tranco, *Tranco List*, Accessed: 2024-12-16, 2024. [Online]. Available: https://tranco-list.eu/.

[21] *MDN Web Docs: Object.freeze()*, Accessed: 2024-12-20, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze.

[22] *MDN Web Docs: Polyfill*, Accessed: 2024-12-11, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.

[23] Díaz, Santiago, *Prototype Pollution Mitigation / Symbol.proto*, Accessed: 2024-11-19, 2023. [Online]. Available: https://github.com/tc39/proposal-symbol-proto/blob/4a181de59786af3ceddb5b595b93534a9a9a76b6/README.md.