Exploiting Input Sanitization for Regex Denial of Service

Efe Barlas*
Purdue University
West Lafayette, Indiana, USA
ebarlas@purdue.edu

Xin Du*
Purdue University
West Lafayette, Indiana, USA
du201@purdue.edu

James C. Davis
Purdue University
West Lafayette, Indiana, USA
davisjam@purdue.edu

ABSTRACT

Web services use server-side input sanitization to guard against harmful input. Some web services publish their sanitization logic to make their client interface more usable, *e.g.,* allowing clients to debug invalid requests locally. However, this usability practice poses a security risk. Specifically, services may share the regexes they use to sanitize input strings — and regex-based denial of service (ReDoS) is an emerging threat. Although prominent service outages caused by ReDoS have spurred interest in this topic, we know little about the degree to which live web services are vulnerable to ReDoS.

In this paper, we conduct the first black-box study measuring the extent of ReDoS vulnerabilities in live web services. We apply the *Consistent Sanitization Assumption*: that client-side sanitization logic, including regexes, is consistent with the sanitization logic on the server-side. We identify a service's regex-based input sanitization in its HTML forms or its API, find vulnerable regexes among these regexes, craft ReDoS probes, and pinpoint vulnerabilities. We analyzed the HTML forms of 1,000 services and the APIs of 475 services. Of these, 355 services publish regexes; 17 services publish unsafe regexes; and 6 services are vulnerable to ReDoS through their APIs (6 domains; 15 subdomains). Both Microsoft and Amazon Web Services patched their web services as a result of our disclosure. Since these vulnerabilities were from API specifications, not HTML forms, we proposed a ReDoS defense for a popular API validation library, and our patch has been merged. To summarize: in client-visible sanitization logic, some web services advertise ReDoS vulnerabilities in plain sight. Our results motivate short-term patches and long-term fundamental solutions.

"Make measurable what cannot be measured." –Galileo Galilei

CCS CONCEPTS

• **Security and privacy → Denial-of-service attacks**; *Web application security*; • **General and reference → Empirical studies**; **Measurement**; **Validation**.

KEYWORDS

Empirical software engineering, regular expressions, ReDoS, web security, denial of service, algorithmic complexity attacks

*Both authors contributed equally to this research.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510047

ACM Reference Format:
Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting Input Sanitization for Regex Denial of Service. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510047

1 INTRODUCTION

Internet-based web services play a major role in modern society. By their nature, web services are accessible through an interface, and so they must handle input from users both legitimate and adversarial. Web services interpret string-based inputs into appropriate types such as email addresses, phone numbers, and credit card information. A common first line of defense is therefore to filter for reasonable-looking input. If this input sanitization is flawed, the health of the web service can be compromised [73].

Unfortunately, a common input sanitization strategy exposes web services to a denial of service attack called Regular expression Denial of Service (ReDoS). Many software systems rely on regular expressions (regexes) for input sanitization [32, 47]. Some of these regexes are *problematically ambiguous* [30, 111] and may require super-linear time (in the input length) to evaluate in an unsafe regex engine [102]. At present, most regex engines are unsafe in this regard [40, 46]. The cost of regex processing, combined with the use of regexes across the system stack, can affect the availability of web services, leading to regex-based denial of service (ReDoS) [42, 43].

The ReDoS problem has been considered from several perspectives. Theoretically, the properties of problematic regexes under different search models have been established, including both Kleene-regular semantics [89, 111, 114] and extended semantics [77]. In terms of the supply chain, Davis *et al.* showed that up to 10% of the regexes in open-source modules are problematically ambiguous [45–47]. With respect to live services, Wüstholz *et al.* showed that problematic regexes in many Java applications are exploitable [114], and Staicu & Pradel showed that 10% of the Node.js-based web services they examined were vulnerable to ReDoS [99]. However, these approaches relied on implementation knowledge; they could not be applied to an arbitrary web service. Prior researchers have not studied whether attackers can identify ReDoS vulnerabilities in a black-box manner. If so, the engineering community should prioritize adopting ReDoS mitigations [48, 93, 104].

In this paper, we describe the first black-box measurement methodology for ReDoS vulnerabilities (§4). We exploit software engineering practice, examining a previously-unstudied source of ReDoS information: the regexes that web services provide for use in client-side sanitization. In the first step of our method, we collect the regexes provided in HTML forms (sampling popular websites) and in APIs (using a directory of services with OpenAPI specifications). Then, we analyze them locally for problematic worst-case behavior in a typical unsafe regex engine. Finally, we ethically probe web services for ReDoS vulnerabilities.

(a) Sanitization in HTML form. Regexes can be applied using an input field's `pattern` attribute, and via JavaScript event handlers.

```yaml
paths:
  /information:
    post:
      parameters:
      - name: x
        in: query
        required: true
        schema:
          type: string
          pattern: '([0-9]+(-?))+'
          minLength: 5
      responses:
        '200':
          description: ...
```

(b) Sanitization in OpenAPI-based web API. Regexes are supported to encode string constraints in popular web API schema definition languages including OpenAPI [9], RAML [5], and API Blueprint [1].

Figure 1: The use of regexes for client-side sanitization in a web form and a web API.

Our findings indicate that emerging software engineering practices on the web expose web service providers to ReDoS (§5). Based on a sample of $N = 1000$ popular websites, we report that web service providers do not reveal ReDoS vulnerabilities through their traditional HTML forms. In contrast, web service providers reveal ReDoS vulnerabilities through API specification documents. In our study of a live web services with OpenAPI specifications: there were 475 web domains; 83 of them document their input sanitization regexes; and 6 of these (15 distinct subdomains) are vulnerable to ReDoS. Web service providers publish much more information about their sanitization practices in their API specifications than in their traditional HTML forms. To summarize our contributions:

- We use the Consistent Sanitization Assumption to design the first black-box ReDoS measurement scheme for web services (§4).
- We identify ReDoS vulnerabilities in several live web services (§5). Comparing traditional HTML forms with the emerging approach of API specification, we report that current API specification practices expose web service providers to ReDoS.
- We describe the responses of engineering practitioners to the vulnerabilities we identified (§5.4). We contribute to the engineering practitioner community through a pull request to a major web API input sanitization library. The pull request has been merged and released. [1]
- We share a dataset of web sanitization regexes to complement existing regex datasets mined from other sources.

Significance: Three aspects of our research contributions are significant. First, we establish the first black-box measurement methodology for ReDoS vulnerabilities in live web services. Second, we use this methodology to identify insecure software engineering practices that affect a growing area of the web: APIs. Third, we offer an anti-ReDoS patch that will benefit millions of dependent modules in the OpenAPI ecosystem. Measurements drive change.

[1]See https://github.com/ajv-validator/ajv/pull/1684 and https://github.com/ajv-validator/ajv/pull/1828.

2 BACKGROUND

2.1 Web Services and Web Interfaces

A web service is a software component (server) with which a user (client) can communicate over the Internet via a uniform resource identifier [58]. Common examples include the web services offered by YouTube and Amazon. Clients interact with a web service using its interface, which is commonly defined in two ways: (1) a browser-based interface; and (2) an application programming interface (API).

Most web services offer browser-based interfaces within their websites. Websites are built using technologies such as HTML, CSS, and JavaScript, and displayed to clients through a web browser. To make websites responsive, web engineers provide interfaces, such as search boxes and login forms, to websites' users. Those interfaces allow users to send data to websites' servers to process the data and respond if needed. To build such user interfaces, engineers often use HTML forms [10], as depicted in Figure 1a.

While browser-based interfaces target the general public, some web services support an Application Programming Interface (API) for automated interactions. APIs are software interfaces that describe how different pieces of software should communicate with each other. Through an API, a web service provider can give a more formal description of how to interact with the service. This description enables engineers to develop software that interacts with the service programmatically. APIs can be described with an informal text-based document, or with a schema definition language such as OpenAPI [9], RAML [5], or API Blueprint [1] (cf. Figure 1b). API semantics may be explicit or implied, *e.g.,* using the conventional meaning of REST verbs to develop a REST-ful API [59].

By their nature, web services interact with untrusted clients. It is therefore standard engineering practice to sanitize any client input, whether it comes via a browser-based interface or an API [73]. Since the client controls this input, sanitization ought always to be performed as part of the server's logic. However, some web services also publish input sanitization logic to their clients, to reduce network traffic and give clients feedback about invalid requests.

Figure 1 depicts common forms of sanitization published to clients. The HTML form in Figure 1a illustrates the two ways to perform client-side sanitization for HTML forms: HTML-based and JavaScript-based [11]. Using HTML-based form validation, engineers can enforce attributes on various HTML tags in HTML forms. The attribute of interest in this work is the "pattern" attribute, which lets an engineer specify the language of legitimate input. For more sophisticated checks, JavaScript-based validation supports custom client-side validation logic applied on a relevant event such as an attempted form submission. Just like HTML-based form validation, regexes can also be used in JavaScript-based validation to check the validity of an input string. Meanwhile, Figure 1b depicts an OpenAPI-style API definition. Similar to HTML forms, API schema documents may constrain request headers, payload structure, and field types and valid values. These constraints may indicate enumerations, numeric ranges, string lengths, and — of interest in our study — regexes prescribing string input languages.

Client-side sanitization can help legitimate users debug their requests, *e.g.,* via feedback from the web browser or an automatically generated client API driver. However, malicious clients can bypass client-side sanitization and send unsanitized content to the web service, so services must sanitize again on the server side [84].

2.2 Regexes and Regex-based Denial of Service

Our work measures a form of *denial of service* that web services risk as a result of sharing input sanitization regexes with their clients.

Denial of Service Attacks: A denial of service attack consumes the resources of a service so that legitimate access to the service is delayed or prevented [83]. There are many types of denial of service attacks, varying in the resource exhausted and the exhaustion mechanism. For example, attacks might exhaust network resources (*e.g.,* distributed denial of service) or computational resources (*e.g.,* algorithmic complexity attacks [43]).

Regex-based denial of service (ReDoS) is a denial of service attack that exhausts computational resources by exploiting the worst-case time complexity of an unsafe regex engine.

Unsafe Regex Engines: A regex describes a language (a set of strings) [67]. To determine whether a string matches a regex, membership testing is conducted by a system component called a regex engine [60]. Most programming languages embed a custom regex engine for efficient interactions with the programming language's string encoding. These regex engines support diverse features with complex semantics [64]. To reduce implementation and maintenance costs, some regex engine developers chose designs that favor simplicity over safety [48] — they use a predictive parsing algorithm with backtracking [14, 40, 46]. The emphasis on simplicity comes at a cost: this algorithm has high time complexity, polynomial or exponential in the worst case.

The high time complexity of the standard regex engine algorithm is triggered by a problematic combination of a regex and an input string. These regexes are *super-linear*; there are input strings w that incur time complexity super-linear in the length of w. Most regexes in this class are *problematically ambiguous* [15, 89, 111, 114]. Because they are ambiguous, these regexes can match a string in multiple ways. The typical regex engine's backtracking search algorithm will explore all potential matching paths before returning a mismatch. When the number of paths or the cost of each path depends on the length $|w|$, the result can be super-linear time complexity. Figure 2 illustrates an example of exponential behavior. Many researchers have proposed tools to identify regex-input pairs with worst-case polynomial or exponential behavior [27, 30, 77, 87, 89, 94, 100, 101, 111, 114].

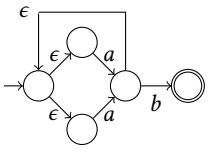

Figure 2: This non-deterministic finite automaton (NFA) corresponds to the regex `(a|a)*b`. Using the typical Spencer algorithm [97], viz. a predictive parse with backtracking, the search space can be exponential in the length of the input. For example, consider the behavior on input "*a*...*a*" (length *k*). The regex does not match this string; the backtracking algorithm explores all 2^k failing paths.

Regex-Based Denial of Service: Regex-based denial of service (ReDoS), exhausts computational resources by exploiting the algorithmic complexity of regex engines [42]. When client-controlled input can trigger worst-case regex behavior (Figure 3), it can be harmful. For example, in 2016 Stack Overflow had a system-wide outage due to ReDoS [57], and in 2019 Cloudflare had a ReDoS outage that affected thousands of its customers [65]. While ReDoS vulnerabilities directly impact compute resources, depending on the system design they may impact higher-order resources. For example, many web services multiplex between clients, *e.g.,* event handler threads [49], and in these designs a ReDoS attack will be more effective. However, even with near-perfect client isolation, *e.g.,* via AWS Lambda, algorithmic complexity attacks like ReDoS provide attackers with an asymmetric attack/defense cost ratio to inflict economic damage [95, 96].

Slow reachable server-side regexes are a security risk. More formally [44], a ReDoS attack requires four *ReDoS Conditions* of a victim web service: (1) It accepts *attacker-controlled input*; (2) It uses a server-side *super-linear regex* on this input; (3) It uses an *unsafe regex engine*; and (4) It has *insufficient mitigations* to insulate other clients from slow server-side regex matches (*e.g.,* timeouts). Although mitigations may reduce the service's ReDoS risk, they do not eliminate it [95, 96]. Therefore, if a service meets Conditions 1–3, we consider it vulnerable to ReDoS.

For example, suppose a web service is built with the Node.js framework. If it has a reachable super-linear regex, then Conditions 1 and 2 are met. The regex may be evaluated on Node.js's unsafe default regex engine (Condition 3). Its slow performance would then affect other clients due to client multiplexing on the Node.js event loop (Condition 4) [49, 85, 99].

2.3 Prior Empirical Studies on ReDoS

The two previous empirical measurements of the extent of ReDoS in practice have used a strong threat model: that the attacker controls the input and also has server-side implementation knowledge.

Under that model, Wüstholz *et al.* identified ReDoS vulnerabilities in open-source Java applications using program reachability analysis, reporting that many Java applications used reachable super-linear regexes [114]. Staicu & Pradel exploited knowledge of the open-source JavaScript software supply chain [90, 113] to predict ReDoS vulnerabilities in web services that use Express, the Node.js server-side framework — 10% of the services they probed had ReDoS vulnerabilities [99].

Although these studies document the risks of ReDoS in practice, their methodologies depend on knowledge of web service internals, and are unsuitable to larger-scale probing of web services in a black-box manner. Finally, studies by Davis *et al.* measured the extent of super-linear regexes within the software module supply chain [45–47], and do not shed light on web service vulnerabilities.

3 ATTACK AND RESEARCH QUESTIONS

Threat Model: Our primary interest is to answer the question: *To what extent do ReDoS vulnerabilities exist in live web services?* As discussed in §2, prior work has measured the possibility of ReDoS (through module analysis) and the presence of ReDoS (through white-box analysis). Thus far we lack a methodology to measure the risk that ReDoS poses to general black-box web services.

We therefore assume the weakest reasonable threat model. First, we suppose the attacker controls only the *input* (Condition 1). Second, we suppose the attacker does *not* have access to the web service's server-side logic. Under this threat model, the primary difficulty is in identifying a reachable super-linear regex to satisfy Condition 2.[2] Once such a regex is identified, the attacker can tailor their input to the regex, then use probes to experimentally determine whether Conditions 3 and 4 are met.

When engineering teams evaluate their own services, this threat model may be needlessly restrictive. But it may imitate the perspective of engineers assessing the risks of incorporating a third-party service or component into their product, or that of adversaries, penetration testers, and "security-scanning-as-a-service" vendors.

Sanitization-Based ReDoS Attacks: Given this constraint, we propose *sanitization-based ReDoS attacks*. As noted in §2, while web services do not typically publish their server-side implementations, some of them do publish client-side input sanitization logic. We adopt the *Consistent Sanitization Assumption* (Figure 3): following engineering conventions, the client-side sanitization logic that a web service publishes is a subset of its server-side sanitization logic. This assumption implies that a super-linear regex used in client-side sanitization logic will fulfill ReDoS conditions 1 and 2: this regex will be applied to attacker-controlled input on the server-side. If true, ReDoS vulnerabilities can be discovered by finding super-linear sanitization regexes in client-side sanitization logic, and then probing web services to test the remaining conditions.

Research Questions: We conduct the first black-box web measurement study of the extent of ReDoS-vulnerable web services in practice. Our operationalized research questions are:

RQ1: How common is regex-based client-side input sanitization? (ReDoS Condition 1)

[2]In the 2000s there were many CVEs for web services that allowed users to specify a regex to be evaluated on the server side (Condition 2), and they used an unsafe regex engine. Such CVEs are now rare, so we suppose a weaker threat model.

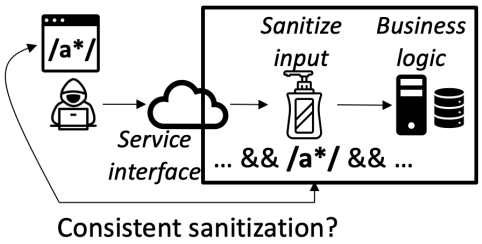

Figure 3: Web service model. For ReDoS, the attacker identifies a web service with a vulnerable sanitization regex, then transmits data that (a) passes preceding constraints, then (b) triggers the worst-case behavior of the regex engine.

RQ2: What proportion of these regexes would be super-linear in an unsafe regex engine? (ReDoS Condition 2)

RQ3: To what extent do these regexes exhibit super-linear behavior in live web services? (ReDoS Condition 3)

RQ4: How does the web service community mitigate these problematic super-linear regexes? (ReDoS Condition 4)

Research ethics constrain us from fully characterizing the extent of ReDoS vulnerabilities with this method. Using a black-box approach, we can measure whether a service meets the first three ReDoS Conditions: (1) it evaluates untrusted input, (2) on a problematic regex, (3) with a slow regex engine. However, we cannot comment on (4) the mitigations the service has in place, whether architectural or runtime. Web services are opaque. Their mitigations are difficult to assess without launching a denial of service attack. In addition, because our method relies only on publicly-accessible service information, we cannot comment on the existence of hidden ReDoS vulnerabilities such as those identified by Staicu & Pradel via implementation inference [99]. With these caveats, our method lets us measure black-box web services for a new perspective on the risks of ReDoS in the wild.

4 METHODOLOGY

Our methodology is shown in Figure 4. Given a web interface, first we analyze its client-side input sanitization logic to determine the input fields and the regexes applied to them (§4.2). Next, we identify any super-linear regexes (§4.3). Then, we ethically probe the web service for ReDoS vulnerabilities (§4.4). Finally, we interact with web service engineers to understand their perspectives (§4.5).

4.1 Web Service Selection

As discussed in §2, web services commonly offer two kinds of interfaces: HTML forms and APIs. We measure ReDoS vulnerabilities through both kinds of interfaces.

HTML Form Interfaces: For HTML form interfaces, we examine the forms within a random sample of 1,000 domains from the top 1 million domains according to the Tranco Top 1M ranking [88]. The Tranco directory is a website popularity ranking designed to address shortcomings of the Alexa list [6].

APIs: API-based interfaces are less common than HTML form interfaces. To obtain sufficient data, we focused on the popular OpenAPI [9] schema description language for REST-ful [59] APIs. We determined popularity using web searches and GitHub stars [29];

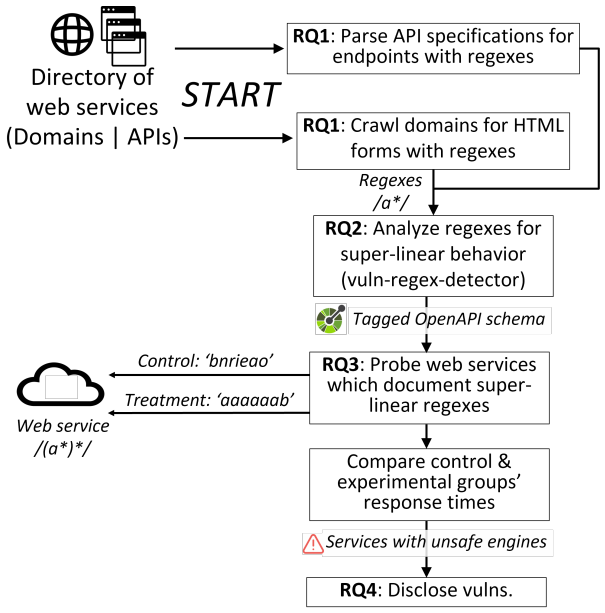

Figure 4: Overview of the study design.

OpenAPI has 23K stargazers, about three times more than the next-most-common API language, API Blueprint [1].

Following prior work on specification measurement [112], we used the `apis.guru` directory [7] to obtain OpenAPI documents.

4.2 RQ1: Input Sanitization Regexes

In this part of the study, our goal is to measure how frequently web services include regexes as part of their client-side input sanitization. To do this, we build a list of live web services and the regexes they use in their client-visible input sanitization. As depicted in Figure 1, software engineers can impose similar input sanitization using the two interface types. The mechanism for specifying this sanitization differs by interface type.

HTML Form Interfaces: For HTML forms, regexes may occur in two places (Table 1). First, in the relevant form field, the string language of valid input can be described by a JavaScript-dialect regex using the HTML5 `pattern` attribute. Second, JavaScript logic can be applied to form fields, *e.g.,* on data entry or button press events. This logic can impose constraints including regex tests.

Table 1: Regex-based sanitization in browser interfaces [81]. Figure 1a also shows both types of HTML Form regexes.

Type	Functions
HTML attribute	`pattern` attribute
JS: String JS: RegExp	String.{match, matchAll, search} RegExp.{test, exec}

The HTML forms that comprise a web service's browser interface may occur on any page. We used the Apify web crawler [12] to crawl each website from its homepage and identify forms. To balance our desire for detailed crawls with the need to not take resources from real users, we used a maximum crawl depth of 500 and fully crawled 66% of the crawled web sites.

After identifying each form, we determine the regexes it uses in its client-side sanitization. We define this set as: "*any regex that is applied to any form field prior to sending form content to the server*". We statically extract the regexes given as form attributes. We use a simple dynamic taint analysis to identify regex constraints in JavaScript logic. First, we monkey-patch the client-side JavaScript regex functions (Table 1) to log each regex-string pair. This is done by modifying those functions' definitions in the browser so that, each time the functions are called, we have access to their input parameters. Then, we drive a web browser via OpenWPM [80], which is a software that can control browsers programmatically, to populate form fields with unique values and simulate a button press. The forms and buttons are detected by parsing the HTML code of each webpage. We use a proxy to discard the resulting form traffic so that we do not spam the web service. Then, inspecting the monkey patch traces, we identify the regexes applied to each form field. This may be a subset of the desired set — our approach omits any regexes that are applied to the substrings of form fields, *e.g.,* logic that splits an email and checks a property on the username.

Unsatisfactory form field values may lead the browser to reject our form *before* our program instrumentation is triggered. To reduce these cases, we solve the constraints encoded within HTML form attributes, *e.g.,* integer constraints directly and regex constraints using Z3 [50], although JavaScript-based constraints may still fail.

APIs: For API-based interfaces we have the same goal: to identify the set of regexes that constrain client input. For such an interface, client input can appear in HTTP headers, endpoints, query strings, and request bodies. In typical API schema definition languages, including OpenAPI, engineers can set regex-based constraints on string inputs. We parse a schema and identify the regex(es) that constrain any string inputs referred to by at least one request schema.

4.3 RQ2: Super-Linear Sanitization Regexes

Our next goal is to measure the proportion of client-side input sanitization regexes that present a *potential* ReDoS vector. We lack knowledge of a web service's server-side implementation, including its choice of regex engine. A regex is a potential ReDoS vector if it has super-linear worst-case behavior in *some* regex engine.

To identify a super-linear regex, we apply the ensemble of state-of-the-art super-linear regex analyses supported by Davis *et al.*'s tool, `vuln-regex-detector` [3, 46]. These analyses vary in their soundness and completeness, so we dynamically test any potentially super-linear regex in a representative unsafe regex engine. Davis *et al.* found the Java, JavaScript, and Python regex engines were in the most unsafe class of engines [46]. Although the Java [106] and JavaScript-V8 [28] regex engines have recently been optimized, the Python regex engine has not. We therefore tested regexes in the Python regex engine (Python v3.8.10).

We define a regex as super-linear using the definition from algorithmic complexity theory [39]: when it exhibits a more-than-linear increase in match time in the input length $|w|$ as we increase the number of "pumps" of the attack input strings. We further distinguished the degree as "high-complexity" and "low-complexity"

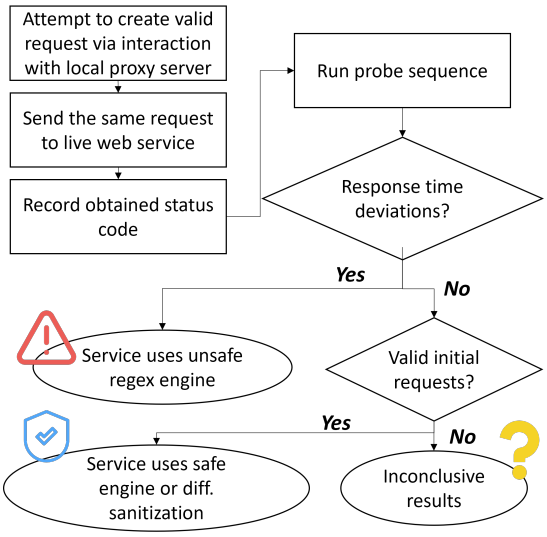

Figure 5: Measurement process for ReDoS Condition 3. We begin with a web service interface, a client input field, and a super-linear regex applied to that input on the client side. As a reachability check, we seek a valid status code from a proxy and then from the live web service. After a probe sequence of treatment and control requests, a decision tree follows.

depending on the number of pumps necessary to yield substantial matching times, similar to Davis *et al.* [46].

4.4 RQ3: Use of Unsafe Regex Engine

By now we have identified interfaces in live web services that publish a super-linear regex used on client input. Our next goal is to understand the proportion of these regexes that are *actual* ReDoS vectors, *i.e.,* testing whether the server uses these regexes in an unsafe regex engine.

4.4.1 Measurement algorithm. As depicted in Figure 5, our measurement algorithm attempts to (1) reach the relevant logic in the interface's server-side implementation; and then (2) identify linear vs. super-linear regex behavior on the server-side while (3) avoiding actually conducting a denial of service attack.

We assume that these live web services follow standard HTTP semantics [59], in particular that if a web service responds to a client request with a success return code (`2XX`) then the request was legitimate. We assume such a request has passed all server-side sanitization. If the Consistent Sanitization Assumption holds, then this means that any client-side regexes were also applied to the relevant input field(s) on the server side. Requests can then be sent to determine whether these regexes exhibit super-linear behavior.

It is possible that the target regex can be reached even without a successful baseline request, but it depends on the cause of the failure. For example, the target regex constraint might be applied *before* the failing condition. Thus, if no successful request can be crafted, super-linear behavior may still be observed; but if we observe linear-time behavior, the results are inconclusive. If we cannot identify a valid request, we use an invalid one.

4.4.2 Crafting a valid client request: **HTML Form Interfaces:** For HTML form interfaces, we identified and satisfied the constraints embedded in HTML form attributes as part of RQ1 (§4.2). To test constraint validity for these interfaces, if a request reaches our HTTP proxy, then we conclude that it passed the client-side sanitization. We send requests to the web service using the Python `requests` [92] module.

APIs: For APIs, the first problem is reaching the target endpoint. For example, reaching an endpoint like `/home/{USER_ID}/photos` requires dynamic information (a `USER_ID`) obtained from the web service. After this, we must satisfy the constraints associated with the fields of the endpoint in question — among other limitations, RESTler's fuzzing strategy may not satisfy the regex constraint that is present for these endpoints. For reachability, we implemented our API analysis as a "checker" plug-in within Microsoft's state-of-the-art REST API fuzzer, RESTler, version v7.4.0 [18]. RESTler uses API conventions to determine dependency relationships between endpoints, with a simple fuzzing dictionary to attempt to satisfy each endpoint's constraints. Once RESTler reaches the target endpoint, our plug-in is called to populate the request fields. We use `json-schema-faker` [8] for this purpose, and then populate the ReDoS-relevant field in the endpoint of interest.

To test constraint satisfaction for APIs using RESTler with our plug-in, we use the Prism mock server tool v4.2.6 [4] to validate requests and generate mock responses according to the OpenAPI specification of interest. Prism returns codes in the `4XX` range if any constraints are missing. We treat other codes as an indicator of satisfied constraints.

4.4.3 Ethical ReDoS probing. We send probe requests by injecting *probe strings* into a previously-sent valid request, or an invalid request if no valid status codes were obtained during the probing experiment. These probe strings are assembled from templates produced by the regex analysis component. The templates contain three strings: a prefix, suffix, and a pump. A probe string is a concatenation of the prefix, one or more repetitions of the pump, and the suffix, and triggers the worst-case behavior of a regex during a mismatch. Each additional pump increases the match time super-linearly in an unsafe regex engine.

We devise a five-stage probing experiment based on that of Staicu & Pradel [99]. Our overall goal is to identify "treatment" input strings that yield a ≥ 1 second increase in response time relative to a comparable "control" string — without causing substantial slowdowns for normal clients. To that end: (1) We identify an initial set of treatment input strings with a range of matching times (200ms to 3s) using the performance of the "maximally unsafe" Python regex engine on our workstation. These input strings should yield a small but measurable time difference in an unsafe regex engine. (2) We send 3 (preferably valid) warm-up requests to address response time noise caused by first-time operations, such as cache filling. These use a valid request if we identified one, else an invalid one. (3) For each timing configuration, we send an *experiment* sequence of 5 requests with the vulnerable field populated with a probe string, and a *control* sequence of 5 requests with that field populated with randomly generated strings of the same lengths (these run in linear time). (4) Both groups of requests are expected to fail at the same

stage of validation, viz. the regex constraint. If the median round-trip response time in the treatment group is substantially larger than in the control group, we conclude that the web service being probed uses an unsafe regex engine. Specifically, we look for a 1-second increase in the median round-trip time for the treatment group. (5) If the service is using an unsafe regex engine, its server hardware or runtime timeouts may affect the actual response time relative to our predication. If we observe a `5XX` response code or a response time greater than 5 seconds, we halt the experiment to avoid harm and consider the regex engine unsafe. Conversely, if the treatment group exhibits deviations but below the 1-second threshold, we manually explore a longer probe sequence.

We identified three known mitigations that can mask unsafe regex engine behavior under this protocol. First, server-side rate limiting could delay our probes regardless of their content. Although rate limiting would presumably not cause the treatment-control *deviations* that we measure, we sent no more than 1 request per second to account for this possibility. Second, caching — either of the validation outcome, or of end-to-end results — could cause only the initial query at each probe size to be slow. We manually observed one case of this form. Third, a recent approach can identify the signatures of anomalously slow regex input [21], although we are not aware of applications of this technique in practice.

4.5 RQ4: ReDoS Mitigation

In this part of the study, our goal was to understand the perspective of the web service engineering community on the use of super-linear regexes in server-side input validation. We assessed this constructively (proposed mitigation), as well as in a responsibly destructive manner (vulnerability identification and disclosure).

Constructively, we assessed the state of ReDoS mitigations in OpenAPI-based automatic client sanitization libraries. We found an absence of mitigations, proposed one, and report on our findings.

"Destructively", we contacted the owners of live web services for which ReDoS Conditions 1–3 held: cases where our experiments identified super-linear regex performance in live web services. Since ReDoS is a security problem, we disclosed such issues to web service engineers using their documented route, *e.g.,* the `security@domain.com` email for major companies. We informed them of a super-linear regex in their client-side input sanitization, presented the attack format, and gave a minimal example. We asked whether they considered this a security vulnerability in their service, and what mitigations they had in place.

4.6 Automating RQ1–RQ3

We automated most parts of this measurement process, using existing tools as indicated. We manually intervened when this automation failed. This was particularly notable for the APIs; these services vary in the accuracy of the semantics that they encode in the OpenAPI schema. We intervened to repair schema syntax, authenticate, and supply values RESTler could not obtain (*e.g.,* some resource ID values or under-documented constraints). Some interventions were guided by a service's error messages.

For one web service with a particularly complex API, we used the official client SDK, documentation, and browser interface to craft valid requests to endpoints with super-linear regexes.

5 RESULTS AND ANALYSIS

For security measurement purposes, we are interested in understanding the extent to which a given *web service* is potentially vulnerable to ReDoS attacks. The attack surfaces in question are clear — individual HTML forms and API endpoints. However, services may employ the same sanitization policy across multiple surfaces. We present results aggregated by web domain, as well as aggregated by subdomains where appropriate.

5.1 RQ1: Published Sanitization Information

Finding 1: Web services frequently use regexes to sanitize input on the client side. 272 of the 696 reachable HTML form domains do so, as do 83 of the 475 studied API domains.

Table 2: Use of regexes in client-side input sanitization. *Domains and sub-entities*: For HTML forms, we report the number of web domains and web pages that apply client-side regexes to any form fields. For APIs, we report by domain and subdomain.

Interface type	# Domains	# Sub-entities
HTML form	272 (39.1%)	30895 (20.6%)
API	83 (17.4%)	322 (30.4%)

HTML Forms: We crawled 1,000 domains sampled randomly from the Tranco Top 1M list. Through web crawling, we found at least one web page for 696 of those web sites. Our crawler failed on the remainder, *e.g.,* blocked by the service, and we omit them from the following statistics. Among the crawled 696 domains, the median number of pages per domain was 104, and the median number of forms per domain was 33.

APIs: We obtained 2231 documents from apis.guru. 549 documents contained at least one operation with a regex validation constraint. These documents corresponded to 83 web services with unique domain names, out of 475 web services. The median number of documents per domain and per subdomain are 1.

Analysis: We observed substantial variation in the number of *unique* regexes amongst the input sanitization regexes. For HTML form's pattern attribute regexes, there were 4966 total regex uses but only 33 unique regexes. Substantial regex re-use across websites is consistent with the findings of Hodován *et al.* [66], who examined the regexes parsed during browsing sessions. This repetition may be the result of client-side library or framework re-use, with the regexes originating in web frameworks or JavaScript libraries rather than independently authored by many engineers. In marked contrast to the duplication of regexes in HTML forms, in the API documents there were 2681 total regex uses and 1841 unique regexes.

For HTML forms, we also note that most regexes were employed in JavaScript logic which was used by 265 domains. Only 31 of domains used the HTML5 `pattern` attribute in any form.

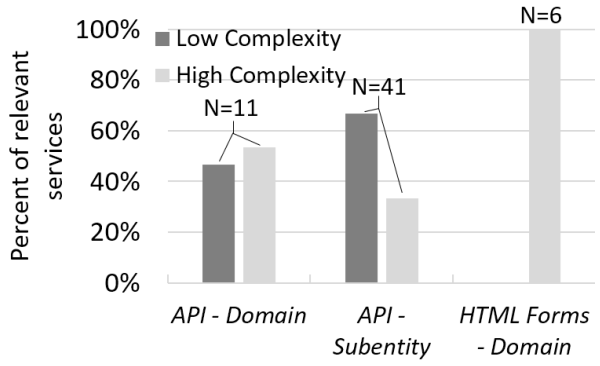

Figure 6: Among the web services with super-linear client-side regexes: the percent of web services with any low- and any high-complexity regexes. N refers to the total number of web services indicated by the lines pointing to the columns.

5.2 RQ2: Super-Linear Regexes

Finding 2: Super-linear regex usage varies widely by interface type. Among the 272 domains with regexes in their HTML forms, only 6 (2%) use a super-linear regex. Meanwhile, among the 83 regex-using API domains, 15 domains (13%) use a super-linear regex in at least one constraint.

HTML Forms: We identified 6 super-linear regexes on 6 distinct web services. Each vulnerable regex appears on exactly one service. Three web services had a super-linear regex on one page each, and the other three used a super-linear regex on 2, 53, and 106 distinct pages correspondingly.

APIs: We found super-linear regexes on documents associated with 15 domains, spanning 45 subdomains.

Analysis: Since high-complexity regexes are more severe than low-complexity regexes, these service providers are exposed to different degrees of risk. Figure 6 shows the distribution of super-linear regexes by time complexity, grouped by interface type.

5.3 RQ3: Live Unsafe Regex Engines

Finding 3: The presence of ReDoS vulnerabilities varies widely by interface type. Our black-box methodology did not identify any ReDoS vulnerabilities from our analysis of HTML forms. From the API analysis, we identified 6 domains (15 subdomains) that meet ReDoS Conditions 1–3: they apply untrusted input to a super-linear regex in an unsafe regex engine on the server side.

Despite our automation, the probing experiments required substantial manual intervention. We chose to consider two kinds of equivalence classes: domains and subdomains. Following the algorithm from Figure 5, once we reached a conclusive result in one of these equivalence classes, we did not attempt other possibilities within the class. We did this in two distinct phases — once at the level of domains, and once at the level of subdomains.

We probed each domain and subdomain aiming to reach a conclusive result for some super-linear regex in their interface. We began with 15 candidate domains (45 subdomains). We identified zero ReDoS-vulnerable domains via HTML form analysis, and six ReDoS-vulnerable domains through API analysis.

Table 3 summarizes our findings for the APIs with super-linear client-side regexes. We reached a conclusive outcome (safe or unsafe regex engine) on at least one probing experiment from 10 domains (32 subdomains). The remaining five domains either did not respond to requests, or required a paid subscription. On 5 domains (12 subdomains), at least one of our probe experiments were inconclusive.

We measured response time deviations (*i.e.,* ReDoS vulnerabilities) for at least one super-linear regex in 6 domains (15 subdomains). Notably, 2 of these domains are on the Tranco Top 1000 list and are major tech companies.

Following our algorithm, we concluded that there were 16 safe services by subdomain — these subdomains have at least one super-linear client-side regex but we did not measure response time deviations on the server side. However, these subdomains all belong to domain *A*, which also has subdomains which had measurable response time deviations. A large company could have distinct policies at the organizational level [38], which may manifest by subdomain as we observed.

Length-Based Mitigation: Davis *et al.* reported that input length checks are a common safety measure for low-complexity regexes [45]. We observed this during our experiments. The *A* domain had many subdomains whose only super-linear regexes were low-complexity. Of the 13 conclusively safe subdomains of *A* in Table 3, 11 used only low-complexity regexes. The error messages from these subdomains indicated that our probe strings were too long.

Table 3: Findings from our study of the APIs with super-linear regexes in their client-visible sanitization logic. Columns represent anonymized domains. Service provider *A* has subdomains with varying properties. We measured response time deviations in 6 domains (15 subdomains). We did not attempt to probe web services which had response time deviations indicating ReDoS caused by high time complexity regexes. *SL*: super-linear. *N/A*: Domain does not have a regex of this type.

Metric	*A*	*B*	*C*	*D*	*E*	*F*	*G*
Number of subdomains	31	1	1	1	1	1	1
Subdomains with SL behavior	10	1	1	0	1	1	1
Subdomains with high-complexity SL behavior	6	N/A	1	Failed experiment	1	1	1
Subdomains with low-complexity SL behavior	5	1	Did not attempt	0	N/A	N/A	N/A
Conclusively safe subdomains	13	0	0	0	0	0	0

5.4 RQ4: ReDoS Mitigation

Finding 4: The maintainers of OpenAPI middleware tools are concerned about ReDoS and interested in eliminating this possibility. It is unclear whether individual web service providers consider the vulnerability a threat.

5.4.1 ReDoS Mitigation for OpenAPI. All of the ReDoS vulnerabilities we identified through our black-box methodology came from APIs, not web forms. We therefore investigated a mitigation for the OpenAPI ecosystem. One benefit of API specifications is that client- and server-side code can be generated automatically. In OpenAPI, two popular code generation tools are `swagger-codegen` and `openapi-generator`, in use by dozens of companies (Table 4). Among other features, these tools can generate server stubs, with input validation code followed by a "fill in the blank" for the business logic. *Their generated code — which includes regex checks — can expose their dependents to ReDoS.*

Table 4: Popular code generators and input validation tools for OpenAPI-based APIs. Data as of February 2022. GH Stars: the number of GitHub stars.

Name	# GH Stars [29]	# Contributors
swagger-codegen	14K	1K
openapi-generator	11.2K	2K

These tools do not address the risk of ReDoS for their dependents. Their documentation does not discuss how the code for regex patterns is generated. According to our tests, these tools generate code in the target programming language and use the default (often unsafe [46]) regex engine in that language. One tool's documentation mentions the risk of ReDoS, but places the burden on the specification engineer to avoid or mitigate such regexes. They do not allow users to tune this logic, *e.g.,* to choose a safe regex engine.

To eliminate this risk, we proposed a patch to the `Ajv` tool. `Ajv` is used by openapi-generator to validate requests, and has several million dependent packages according to GitHub.

Our goal was to allow software engineers to choose the safe regex engine `RE2` [41] instead of the built-in programming language regex engine. `Ajv` is sometimes used in client-side contexts, so the engineering team prioritizes a small binary. Adding the Node.js bindings for `RE2` in `Ajv`'s dependencies would more than double `Ajv`'s unpacked binary size, from 1.02MB to 2.31MB. Our patch therefore used the Factory pattern [61], allowing users to inject another regex engine as a dependency at runtime. The `Ajv` engineering team reviewed our patch and included it in release v8.8, along with documentation outlining how the patch should be used to eliminate risk of ReDoS caused by input sanitization. Our patch will allow the millions of `Ajv` dependents to eliminate this form of ReDoS in their applications.

5.4.2 Responses to ReDoS Vulnerability Disclosures. As described in §4.5, we disclosed possible ReDoS vulnerabilities to live web service providers who met ReDoS conditions 1-3.

To summarize the responses: (1) Four service providers did not respond; (2) One major technology company acknowledged and repaired the disclosed vulnerability; (3) One major technology company initially told us that they did not perceive a vulnerability, but after several months have informed us that they have patched the unsafe regexes. Ultimately, both Microsoft and Amazon Web Services made changes to repair their unsafe regexes. Overall, the sample size is too small for comment.

6 DISCUSSION

Should web service providers prioritize ReDoS mitigations? Several research communities have investigated the ReDoS problem, including from empirical software engineering [45–47], systems [41, 49, 85], cybersecurity [48**?**], and theory [89, 93, 111, 114]. This research investment has somewhat shaky motivation: Crosby's proposal of ReDoS [42], case studies of regex-induced service outages [57, 65], and three empirical measurement studies [45, 99, 114]. Our study provides a new perspective: large-scale black-box measurements of ReDoS risks using a weak threat model (§5.2, §5.3). Our findings establish the first systematic and empirical evaluation of ReDoS risks of web services, without an assumption of their frameworks.

Our measurements indicate that many web services are safe from ReDoS under this threat model. In particular, for web services whose interfaces are traditional HTML forms, few sanitization regexes are revealed on the client side, and these regexes are not superlinear (§5.1, §5.2). In contrast, web services that publish APIs face more risk of ReDoS. In publishing API specifications, web services are choosing to reveal more about their server-side sanitization logic. However, the cause is unclear. We conjecture two explanations for further examination. First, the choice may be deliberate. Software engineers may be providing a fuller definition of their input validation constructs in their API specifications, so that code generation tools can be used to automatically handle input validation. Alternatively, it may be accidental. Software engineers may be using tools which generate API specifications from code (similar to model extraction), rather than writing specifications first and then generating code from them. This process may be inadvertently exposing internally-used regexes, which could explain the greater regex variety and greater incidence of ReDoS among API regexes.

A visibility-security tradeoff: Our measurements indicate a tradeoff between visibility and security (§5.3). Web service providers who promote usability by specifying the nature of valid input may expose themselves to ReDoS. Although this class of attacks could be mitigated by hiding the sanitization rules, software engineers should not seek security through obscurity [13]. Indeed, describing the characteristics of valid input is necessary to enable communication. Software engineers should not need to choose between visibility and security.

Rather than obscuring input formats, the software engineering community would benefit from principled solutions to ReDoS. Davis *et al.* described several sound solutions, and they concluded that making regex engines safe seemed like the most natural mitigation [48]. Further research into adopting safe regex engines [41, 110] or retrofitting existing unsafe engines [48] will improve the safety of software engineering practice. Middleware for input sanitization can use a level of indirection to select a safe regex engine. In our mitigation study (§4.5), we found that middleware providers were

happy to accept such a change. However, they were concerned with introducing an external dependency on a safe regex engine, indicating that improving the safety of programming language regex engines should be an area of focus for a long-term solution.

Using API specifications has many advantages from a software engineering standpoint. Specific to ReDoS, the standardization they provide allows engineers to use specification-compatible middleware tools. Hence, any security patches to these tools can protect many web services at once. We took advantage of this centralization to provide ReDoS mitigations to the numerous dependents of such middleware tools. The same property that lends itself to easy exploitation also lends itself to a centralized repair.

Mismatch between OpenAPI SDL and needs in practice: Ideally, an interface specification should describe everything necessary for successful communication. However, during our measurements we observed that most OpenAPI documents were underspecified. Our requests, which passed the specified validation constraints according to the Prism tool, were still invalid according to the server.

We identified several causes of these underspecifications. One of these causes is well known — the OpenAPI syntax cannot indicate dependencies between requests. Although RESTler [18] attempts to infer these dependencies using heuristics, it cannot handle all cases. We encountered several more causes during our experiments. Some API documents indicate the full set of possible payload variables, but actually accept subsets of those variables. Some parameters are interdependent/coupled, *e.g.,* including one optional parameter requires including others [2]. Some parameters of "`string`" type are actually type-aliased within the web service, *e.g.,* strings that represent a comma-separated numeric sequence. These various missing semantics are of practical utility, and the maintainers of API specification languages should consider supporting them.

7 RELATED WORK

Our work descends from two lines of research: web service vulnerability scanning, and regular expression engineering.

We employed a black-box probing methodology to scan web services for a specific class of security vulnerability. Other researchers have employed grey-box and white-box methodologies for this vulnerability [99, 114]. Researchers and commercial tools offer black-box and grey-box scanning for diverse vulnerabilities [25, 52, 53], including algorithmic complexity vulnerabilities [31, 72, 87], service crashes [18, 19, 62], cross-site scripting (XSS) [24, 55, 103], and SQL injection [78]. Notably, some researchers have pursued the opposite of our Consistent Sanitization Assumption to identify cases where backend sanitization appears problematically inconsistent [71, 84].

While our study focuses on regex cybersecurity, researchers have considered other aspects of the regex engineering lifecycle. Michael *et al.* reported that many software engineers find regex engineering difficult [79]. To assist the engineering community in this domain, researchers have recently described regex engineering practices related to composition [20], comprehension [33], and testing [109]; identified common regex bug patterns and taxonomies [56, 70, 108]; and proposed tools to support regex comprehension [26], testing [69, 98], and repair [76, 86]. There has also been a longstanding effort to automatically compose regexes, with diverse approaches including formal methods [16, 17, 35, 51, 63, 75], evolutionary algorithms [22, 23, 37], optimization [74, 91], crowdsourcing [36], natural-language translation [34], and human-in-the-loop interactive development [54, 115].

8 THREATS TO VALIDITY

This paper describes a substantial web measurement study covering two distinct interface types. We acknowledge a variety of threats to the validity of our findings, and note mitigating factors.

Construct validity: The primary threat here is in our definition of a browser-based web interface: via HTML forms. While HTML forms appeared in 80% of the web services whose HTML form interfaces we crawled, trends such as Single-Page Applications [82] process user input without form submissions. This is also a threat to external validity, as we cannot comment on the risk of ReDoS for such web services. Beyond this construct, we relied on definitions of super-linear regexes and regex-based denial of service. These concepts are well established in the research literature, and we measured them with state-of-the-art tools and probing methodologies.

Internal validity: Our methodology does not let us measure the degree of a ReDoS vulnerability. We identified super-linear regexes that are applied to user input in an unsafe regex engine on the server side. However, we cannot assess ReDoS Condition 4 (ReDoS mitigations §2), without either having server-side knowledge or launching a full-scale denial of service attack. To shed light on this threat, in §5.4 we discussed perspectives from the web service engineering community.

There are three potential sources of under-reporting in our probing methodology. First, as noted in §4.4, some web services cache end-to-end results, and this caching will mask worst-case behavior when we use identical worst-case probe strings. Second, we conservatively chose input lengths for our probes based on the slowdowns observed on a workstation-grade machine. If a web service provider processes input on a server-class machine, the response time deviation induced by our probes may not be observable. Finally, our decision tree yielded inconclusive results in 12 cases (Figure 5).

External validity: Our goal was to measure the extent of ReDoS vulnerabilities in live web services. The populations we used may have biased our results. We probed web services that were listed in directories of live web services — from the Tranco Top 1M directory [88] for HTML form interfaces, and from `apis.guru` for OpenAPI interfaces. For *HTML forms*, we randomly sampled from the top 1 million web domains for HTML forms. We expect these results to generalize to other popular websites; popularity may be correlated with a certain caliber of engineering (in order to service the client load) and so our results may not generalize to less popular websites. A related bias is that 3.4% of HTML form services rejected our connections outright because we were using VMs from Google Cloud Platform for our experiments. For *APIs*, we considered all API specifications from `apis.guru`. This directory only contains OpenAPI specifications from 475 distinct domains (1,059 distinct subdomains). Different results may emerge from studying other API specification directories, *e.g.,* SwaggerHub or by mining GitHub. We performed a preliminary analysis on SwaggerHub specifications, and found that often they are simpler and are not associated with a live web service URL. Echoing Wittern *et al.* [112], we suggest that

the results we obtained from `apis.guru` may be more representative of *engineered* OpenAPI specifications.

Beyond limitations in our sampling, generalizability is threatened by our black-box methodology. Our approach depends on the *Consistent Sanitization Assumption*: that web services are consistent in their input sanitization, *i.e.,* that client-visible input sanitization is also applied on the back-end. This assumption permits a scalable black-box approach. However, without full knowledge of server-side logic, we may omit server-side regex evaluations that are not exposed to clients. Web services may apply additional or alternative sanitization on the back-end. For example, the ReDoS vulnerabilities identified by Staicu & Pradel [99] would likely not have been discovered using our methodology, since they targeted regexes that would only be used server-side in HTTP header processing. Conversely, client-side sanitization gives us insight into "business logic" regexes that might not be visible through examination only of the open-source software supply chain, as Davis *et al.* [45] and Staicu & Pradel [99] did. Our findings thus complement the prior empirical studies of the risks of ReDoS in practice.

9 FUTURE WORK

Transferring ReDoS vulnerabilities: Software engineers solve similar problems in similar ways [68]. A general version of the Consistent Sanitization Assumption is possible: that *web services validate similar content in similar ways*, so sanitization logic revealed by one service may be transferred to another. For example, suppose two web services use an accessibility feature like ARIA labels [107] to label a form field as an email. If one service provides client-side sanitization logic, similar logic might be in use by the other.

Why are API practices more dangerous? In answering each research question, there were marked differences between ReDoS risks in traditional HTML forms as compared to the emerging approach of API specification. We conjectured two causes (§6): providing detailed API specifications to ease the development of input validation logic, and inadvertent exposure resulting from API extraction from server-side code. We believe this finding bears further investigation.

Improved tooling: Although we chose RESTler to help us reach endpoints in complex APIs, we eventually performed manual intervention for most of the 32 APIs we probed. In practice, API specifications underspecify valid interactions. When we intervened, we consulted API documentation as well as the service error messages. Incorporating NLP techniques into automated API interactions is a natural direction for improved black-box web service testing [62].

Regex dataset: Previous researchers have collected regex datasets from open-source software repositories [32, 47], with applications including improved regex usability tools and safer regex engines [48, 105]. To complement this effort, we contribute a dataset of *web input sanitization* regexes. This dataset contains the ~ 1850 unique regexes identified during our experiments.

10 CONCLUSIONS

Regex-based denial of service (ReDoS) has received much recent attention. Web service providers are curious about the degree to which ReDoS threatens them, and regex engine maintainers wonder whether they should prioritize optimizations to ameliorate ReDoS. In light of this interest, we report the results of the first black-box measurement study of ReDoS vulnerabilities on live web services. Our method is based in the observation that server-side input sanitization may be mirrored on the client-side as part of usability engineering. We therefore examined the extent to which super-linear regexes on the client side can be exploited as ReDoS vulnerabilities on the server side. We compared two common interface types: HTML forms ($N = 1,000$ domains) and APIs ($N = 475$ domains). We report that although client-visible regexes are common in both types of interfaces, super-linear regexes are only common in APIs. We identified ReDoS vulnerabilities in the APIs of 6 domains (15 subdomains) including in services operated by major technology companies. Our findings add weight to the concerns of researchers about the risks of ReDoS in practice. Specifically, we show that the movement toward API specification development provides leverage for ReDoS attacks.

ACKNOWLEDGMENTS

We thank A. Kazerouni and the anonymous referees for their criticisms. Barlas and Du acknowledge support from Purdue University's Summer Undergraduate Research Fellowship program (SURF) and the Purdue University Center for Programming Principles and Software Systems (PurPL).

RESEARCH ETHICS

Our methodology could be construed as conducting denial of service attacks against live web services. Attacking web services is unethical. We did not do so! As discussed in §4, we imitated Staicu & Pradel [99] by using *ReDoS probes* instead of attacks. Prior researchers have contributed theoretical and empirical understanding of worst-case regex performance, enabling us to accurately predict the worst-case performance of a problematic regex. This prediction allows us to size the probes to minimize the risk to the service provider. Our method could introduce a user-perceivable slowdown comparable to a network hiccup, yet still demonstrates the possibility of a ReDoS attack by a malicious actor. We believe this cost is an acceptable price for the data we have collected. However, as a consequence of our ethical probing methodology, we are limited in what we can claim about the vulnerability of web services (§6).

DATA AVAILABILITY

An artifact is available at https://doi.org/10.5281/zenodo.5916441. The artifact has a dataset and our vulnerability identification tools. *Dataset*: We provide a list of regexes found in web forms and API specifications and their analysis reports per `vuln-regex-detector`. We also share the list of web services with regexes in their API specifications, and the list of all web services with API specifications.

Vulnerability identification tools: We share one tool for browser-based interfaces, and one tool for APIs (OpenAPI). The browser-focused tool crawls web services for regex use in web forms and JavaScript files. It produces OpenAPI specifications for endpoints which are using vulnerable regexes in the front-end. The API-focused tool parses OpenAPI specifications for vulnerable regexes and endpoints. It also probes the web services to assess whether each vulnerable regex is exploitable for ReDoS.

REFERENCES

[1] 2014. Documentation | API Blueprint. https://apiblueprint.org/documentation/
[2] 2015. Support interdependencies between query parameters · Issue #256 · OAI/OpenAPI-Specification. https://github.com/OAI/OpenAPI-Specification/issues/256
[3] 2019. GitHub - davisjam/vuln-regex-detector: Detect vulnerable regexes in your project. https://github.com/davisjam/vuln-regex-detector
[4] 2019. Prism | Open-Source HTTP Mock and Proxy Server. https://stoplight.io/open-source/prism
[5] 2020. About RAML. https://raml.org/about-raml
[6] 2021. Alexa – Top sites. https://www.alexa.com/topsites
[7] 2021. Browse APIs - APIs.guru. https://apis.guru/
[8] 2021. json-schema-faker/json-schema-faker. https://github.com/json-schema-faker/json-schema-faker
[9] 2021. OpenAPI Specification Version 3.0.3. https://swagger.io/specification/.
[10] 2022. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form
[11] 2022. https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation
[12] 2022. https://apify.com/
[13] 2022. Security Through Obscurity. https://en.wikipedia.org/wiki/Security_through_obscurity
[14] AV Aho, Monica S Lam, R Sethi, and JD Ullman. 2013. *Compilers: Pearson New International Edition: Principles, Techniques, and Tools.* Pearson.
[15] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. 2008. General Algorithms for Testing the Ambiguity of Finite Automata. In *International Conference on Developments in Language Theory*.
[16] Rene Alquezar and A Sanfeliu. 1999. Incremental Grammatical Inference From Positive and Negative Data Using Unbiased Finite State Automata. (1999).
[17] Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337–350.
[18] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *International Conf. on Software Engineering (ICSE)*.
[19] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. *International Conference on Software Testing, Verification and Validation (ICST)* (2020), 387–397. https://doi.org/10.1109/ICST46399.2020.00046
[20] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. 2019. Exploring tools and strategies used during regular expression composition tasks. In *IEEE International Conference on Program Comprehension (ICPC)*. IEEE.
[21] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In *IEEE Symposium on Security and Privacy (SP)*.
[22] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. 2012. Automatic generation of regular expressions from examples with genetic programming. In *International Conference on Genetic and Evolutionary Computation Companion (GECCO)*. 1477–1478.
[23] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1217–1230.
[24] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular expressions considered harmful in client-side XSS filters. In *The Web Conference (WWW)*.
[25] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the art: Automated black-box web application vulnerability testing. In *IEEE Symposium on Security and Privacy*. 332–345. https://doi.org/10.1109/SP.2010.27
[26] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: Visual Debugging of Regular Expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2591062.2591111
[27] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *EPTCS: Automata and Formal Languages 2014* 151 (2014), 109–123.
[28] Martin Bidlingmaier. 2021. An Additional Non-Backtracking RegExp Engine. https://v8.dev/blog/non-backtracking-regexp
[29] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://linkinghub.elsevier.com/retrieve/pii/S0164121218301961
[30] Claus Brabrand and Jakob G. Thomsen. 2010. Typed and unambiguous pattern matching on strings using regular expressions. *Symposium on Principles and Practice of Declarative Programming (PPDP)* (2010).
[31] Alan Cha, Erik Wittern, Guillaume Baudart, James C. Davis, Louis Mandel, and Jim A. Laredo. 2020. A Principled Approach to GraphQL Query Cost Analysis. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[32] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2931037.2931073
[33] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE)*.
[34] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Programming Language Design and Implementation (PLDI)*. 487–502. https://doi.org/10.1145/3385412.3385988
[35] Nariyoshi Chida and Tachio Terauchi. 2020. Automatic Repair of Vulnerable Regular Expressions. (2020). http://arxiv.org/abs/2010.12450
[36] Robert A. Cochran, Loris D'Antoni, Benjamin Livshits, David Molnar, and Margus Veanes. 2015. Program boosting: Program synthesis via crowd-sourcing. In *Principles of Programming Languages (POPL)*, Vol. 50. 677–688.
[37] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1280–1287.
[38] Melvin E Conway. 1968. How do committees invent. *Datamation* 14, 4 (1968).
[39] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[40] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).
[41] Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html
[42] Scott Crosby. 2003. Denial of service through regular expressions. In *USENIX Security work in progress report*.
[43] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security*.
[44] James C. Davis. 2020. *On the Impact and Defeat of Regular Expression Denial of Service*. Ph.D. Dissertation. Virginia Tech.
[45] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[46] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[47] James C Davis, Daniel Moyer, Ayaan M Kazerouni, and Dongyoon Lee. 2019. Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study. In *Automated Software Engineering (ASE)*.
[48] James C. Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *IEEE Security and Privacy (S&P)*.
[49] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *USENIX Security Symposium (USENIX Security)*.
[50] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[51] François Denis. 2001. Learning regular languages from simple positive examples. *Machine Learning* 44, 1-2 (2001), 37–66.
[52] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security*. 523–538. https://doi.org/10.1007/BF03325089
[53] Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. 2014. Web application testing: A systematic literature review. *Journal of Systems and Software* 91, 1 (2014), 174–201. https://doi.org/10.1016/j.jss.2014.01.010
[54] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Computer-Human Interaction (CHI)*.
[55] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *ACM conference on Data and application security and privacy (CODASPY)*.
[56] Aryaz Eghbali and Michael Pradel. 2020. No Strings Attached : An Empirical Study of String-related Software Bugs. In *Automated Software Engineering (ASE)*.
[57] Stack Exchange. 2016. Outage Postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.
[58] Christopher Ferris and Joel Farrell. 2003. What are web services? *Commun. ACM* 46, 6 (2003), 31.
[59] Roy T Fielding and Richard N Taylor. 2000. Principled design of the modern Web architecture. In *International Conference on Software Engineering (ICSE)*.
[60] Jeffrey EF Friedl. 2002. *Mastering regular expressions*. O'Reilly Media, Inc.
[61] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts.
[62] Patrice Godefroid, Bo Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2020).

[63] E. Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978), 302–320. https://doi.org/10.1016/S0019-9958(78)90562-4

[64] Jan Goyvaerts. 2016. A list of popular tools, utilities and programming languages that provide support for regular expressions, and tips for using them. https://www.regular-expressions.info/tools.html

[65] Graham-Cumming, John. [n.d.]. Details of the Cloudflare outage on July 2, 2019. https://web.archive.org/web/20190712160002/https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[66] Renáta Hodován, Zoltán Herczeg, and Ákos Kiss. 2010. Regular expressions on the web. In *International Symposium on Web Systems Evolution (WSE)*.

[67] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2006. *Automata theory, languages, and computation*. Vol. 24. 19 pages.

[68] John C. Knight and Nancy G. Leveson. [n.d.]. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. SE-12, 1 ([n. d.]), 96–109. https://doi.org/10.1109/TSE.1986.6312924

[69] Eric Larson. 2018. Automatic Checking of Regular Expressions. In *Source Code Analysis and Manipulation (SCAM)*.

[70] Eric Larson and Anna Kirk. 2016. Generating Evil Test Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation (ICST)*. https://doi.org/10.1109/ICST.2016.29

[71] Nuo Li, Tao Xie, Maozhong Jin, and Chao Liu. 2010. Perturbation-based user-input-validation testing of web applications. *Journal of Systems and Software* 83, 11 (2010), 2263–2274. https://doi.org/10.1016/j.jss.2010.07.007

[72] Pengui Li, Yinxi Liu, and Wei Meng. 2021. Understanding and Detecting Performance Bugs in Markdown Compilers. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[73] Xiaowei Li and Yuan Xue. 2014. A Survey on Server-Side Approaches to Securing Web Applications. *ACM Comput. Surv.* 46, 4, Article 54 (March 2014), 29 pages. https://doi.org/10.1145/2541315

[74] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular expression learning for information extraction. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 21–30. https://doi.org/10.3115/1613715.1613719

[75] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2020. TransRegex: Multi-modal Regular Expression Synthesis by Generate-and-Repair. (2020). http://arxiv.org/abs/2012.15489

[76] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2020. FlashRegex: Deducing Anti-ReDoS Regexes from Examples. In *Automated Software Engineering (ASE)*. 659–671.

[77] Yinxi Liu, Mingxue Zhang, and Wei Meng. [n.d.]. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021-05). IEEE.

[78] Michael Martin and Monica S Lam. 2008. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *USENIX Security*.

[79] Louis G Michael IV, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are Hard : Decision-making, Difficulties, and Risks in Programming Regular Expressions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[80] Mozilla. 2021. https://github.com/mozilla/OpenWPM

[81] Microsoft Developer Network. 2021. HTML: HyperText Markup Language. https://developer.mozilla.org/en-US/docs/Web/HTML

[82] Microsoft Developer Network. 2021. SPA (Single-page application). https://developer.mozilla.org/en-US/docs/Glossary/SPA

[83] Michael Nieles, Kelley Dempsey, and Victoria Yan Pillitteri. 2017. *An Introduction to Information Security*. Number NIST SP 800-12r1. https://doi.org/10.6028/NIST.SP.800-12r1

[84] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. 2004. Bypass testing of web applications. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (2004), 187–197. https://doi.org/10.1109/ISSRE.2004.13

[85] Andres Ojamaa and Karl Duuna. 2012. Assessing the security of Node.js platform. In *International Conference for Internet Technology and Secured Transactions*.

[86] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic repair of regular expressions. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3360565

[87] Theoolos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Computer and Communications Security (CCS)*.

[88] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. *Network and Distributed System Security Symposium (NDS))* (2019). arXiv: 1806.01156.

[89] Asiri Rathnayake and Hayo Thielecke. 2014. *Static Analysis for Regular Expression Exponential Runtime via Substructural Logics*. Technical Report.

[90] Eric S. Raymond. 2000. *The Cathedral and the Bazaar*. Number July 1997. 1–35 pages. https://doi.org/10.1007/s12130-999-1026-0

[91] Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding missing words to regular expressions. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.

[92] Kenneth Reitz. 2020. requests: Python HTTP for Humans. https://requests.readthedocs.io

[93] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic regex matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[94] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: Crafting Regular Expression DoS Attacks. In *Automated Software Engineering (ASE)*.

[95] Mor Sides, Anat Bremler-Barr, and Elisha Rosensweig. 2015. Yo-Yo Attack: vulnerability in auto-scaling mechanism. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 103–104.

[96] Gaurav Somani, Manoj Singh Gaur, Dheeraj Sanghi, Mauro Conti, Muttukrishnan Rajarajan, and Rajkumar Buyya. 2017. Combating DDoS attacks in the cloud: requirements, trends, and future directions. *IEEE Cloud Computing* (2017).

[97] Henry Spencer. 1994. A regular-expression matcher. In *Software solutions in C*. 35–71.

[98] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *Workshop on Formal Techniques for Java-like Programs*.

[99] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium (USENIX Security)*.

[100] Satoshi Sugiyama and Yasuhiko Minamide. 2014. Checking Time Linearity of Regular Expression Matching Based on Backtracking. *Information and Media Technologies* 9, 3 (2014), 222–232.

[101] Martin Sulzmann and Kenny Zhuo Ming Lu. 2017. Derivative-Based Diagnosis of Regular Expression Ambiguity. *International Journal of Foundations of Computer Science* 28, 5 (4 2017), 543–561. https://doi.org/10.1142/S0129054117400068

[102] Ken Thompson. 1968. Regular Expression Search Algorithm. *Communications of the ACM (CACM)* (1968).

[103] Omer Tripp, Omri Weisman, and Lotem Guy. 2013. Finding your way in the testing jungle: a learning approach to web security testing. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2483760.2483776

[104] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[105] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[106] Brink van der Merwe, Jacobie Mouton, Steyn van Litsenborgh, and Martin Berglund. 2021. Memoized Regular Expressions. In *International Conference on Implementation and Application of Automata*. Springer, 39–52.

[107] W3C. 2017. Accessible Rich Internet Applications (WAI-ARIA) 1.1. https://www.w3.org/TR/wai-aria

[108] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. 2020. An Empirical Study on Regular Expression Bugs. In *Mining Software Repositories (MSR)*.

[109] Peipei Wang and Kathryn T Stolee. 2018. How well are regular expressions tested in the wild?. In *Foundations of Software Engineering (FSE)*.

[110] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *Networked Systems Design and Implementation (NSDI)*.

[111] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *International Conference on Implementation and Application of Automata*. Springer, 322–334.

[112] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. 2019. An Empirical Study of GraphQL Schemas. In *International Conference on Service-Oriented Computing (ICSOC)*.

[113] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1145/2901739.2901743

[114] Valentin Wüstholz, Oswaldo Olivo, Marijn J H Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[115] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *ACM Symposium on User Interface Software and Technology (UIST)*.