Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions

David Klein*, Thomas Barber†, Souphiane Bensalim†, Ben Stock‡ and Martin Johns*
*Technische Universität Braunschweig {david.klein,m.johns}@tu-braunschweig.de
†SAP Security Research {thomas.barber,souphiane.bensalim}@sap.com
‡CISPA Helmholtz Center for Information Security stock@cispa.de

***Abstract*—Despite the considerable amounts of resources invested into securing the Web, Cross-Site Scripting (XSS) is still widespread. This is especially true for Client-Side XSS as, unlike server-side application frameworks, Web browsers do not ship with standard protection routines, so-called sanitizers. Web developers, therefore, have to either resort to third-party libraries or write their own sanitizers to stop XSS in its tracks. Such custom sanitizer routines – dubbed *hand sanitizers* in the following – are notoriously difficult to implement securely.**

In this paper, we present a technique to automatically detect, extract, analyze, and validate JavaScript sanitizer functions using a combination of taint tracking and symbolic string analysis. While existing work evaluates server-side sanitizers using a small number of applications, we present the first large-scale study of client-side JavaScript sanitizers. Of the most popular $20{,}000$ websites, our method detects 705 unique sanitizers across $1{,}415$ domains, of which 12.5% are insecure. Of the vulnerable sanitizers, we were able to automatically generate circumventing exploits for 51.3% of them, highlighting the dangers of manual sanitization attempts. Interestingly, vulnerable sanitizers are present across the entire range of website rankings considered, and we find that most sanitizers are not generic enough to thwart XSS if used in just a slightly different context.

Finally, we explore the origins of vulnerable sanitizers to motivate adopting a standardized sanitization API available directly in the browser.

1. Introduction

The Web is arguably the most important application delivery platform today. New applications are frequently launched first, or even exclusively, as Web applications. Browsers and the JavaScript language have rapidly gained new features to aid this growth. On the other hand, breaking changes (e.g., removing functionality) are avoided by all major browsers to keep compatibility with old websites intact. This means that the underlying development model has not changed since the advent of JavaScript.

To develop dynamic content on the client, developers have to generate markup with JavaScript which is then added to the DOM. This mix of (static) markup and code is a frequent cause of issues. One of the longest-standing security issues for Web applications is cross-site scripting (XSS). XSS occurs when unfiltered and unsanitized attacker-controllable input is interpreted as code, either because it is insecurely intermixed with HTML or JavaScript. Despite considerable attention from both the research community as well as browser and framework vendors, it is still highly relevant today. In fact, XSS has consistently ranked as one of the most critical security risks for Web applications [1, 2, 3]. The increased usage of JavaScript to provide web application logic has also led to a rise in client-side XSS [4] vulnerabilities [5, 6, 7], in some cases affecting large technology companies such as Google [8] and Facebook [9].

To protect against this class of errors, developers have to filter or sanitize input originating from dangerous sources. Unfortunately, as browsers do not provide standard routines, many developers attempt sanitization through unfit means such as regular expressions. These are unreliable, not only due to the complexity of the HTML5 language itself but also aggravated by the fact that all major browsers use very lenient HTML parsers. This allows even malformed HTML code to render properly, aiding usability but also hindering sanitization efforts and opening the door to abuse from attackers. This is not a new problem; browsers used to ship with regular expression-based XSS protection mechanisms, which were shown to be unreliable [10] and consequently removed. This fact has even reached developer folklore, e.g., the most popular StackOverflow answer on how to match XHTML with regular expressions states that "using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes" [11].

The few notable examples of third-party sanitization libraries, such as Google Closure and DOMPurify, are often used only by security-aware sites, which tend not to construct dangerous flows in the first place. Further, while the Content Security Policy (CSP) is claimed to be "one of the most promising countermeasures against XSS" [12], numerous studies have shown that policies are often not deployed at all or are trivially insecure in 95% of all cases [12, 13, 14, 15]. Hence, hand-written sanitizers are likely here to stay.

In this paper we present a technique to automatically detect, extract, analyze, and validate client-side JavaScript sanitizer functions on a large scale and apply this to the analysis of real-world sanitizers across 20,000 popular sites. To summarize, our main contributions are as follows:

- We develop a mechanism to automatically detect sanitizing functions in real-world JavaScript based on operation traces in collected taint data.
- Based on these operation traces, we present *SemAttack*, an automaton-based framework which can assess sanitizer security and automatically generate

payloads to bypass insecure sanitizers.

- We present, to our knowledge, the first large-scale assessment of JavaScript sanitizer usage and effectiveness in the 20,000 most popular websites.
- Through our empirical analysis, we also highlight common mistakes JavaScript developers make and insights on how they approach such tasks. In light of these findings, we motivate the need for browser-supported sanitization routines.

The accompanying material for this work is available at https://github.com/ias-tubs/hand_sanitizer. Release of the dataset collected during this study is not planned as it would reveal publicly exploitable vulnerabilities which have not necessarily been patched by website owners. Instead, we made a sample dataset (e.g., as described in Section 4.3) illustrating our method available.

2. Technical Background and Related Work

In this section, we present the required technical background for this work, namely client-side XSS vulnerabilities, input sanitization, and the OWASP recommendations on how to encode input to prevent client-side XSS. We also highlight related work throughout this section.

2.1. Client-Side XSS

By default, JavaScript code running in a browser is bound to allow access only to resources from the same origin (the tuple of protocol, host, and port). Hence, an attacker cannot simply build a page that loads a target page in a frame and directly access its content through JavaScript. To achieve this, the attacker's code has to run in the same origin as that of the target page. The class of attack in which the adversary abuses vulnerabilities to inject their code into another page is called Cross-Site Scripting (XSS) and has been known since the early 2000s [16]. The goal of an attacker is to inject JavaScript code (either directly or within HTML markup) to execute some nefarious action, such as stealing cookies or phishing credentials. One specific subclass of XSS is Client-Side Cross-Site Scripting (referred to as DOM-based XSS when it was first discovered in 2005 [4]). Here, the *client-side* notion refers to the fact that the insecurity lies within client-side JavaScript rather than server-side code.

Client-side XSS flaws can occur when attacker-controllable data is used within dangerous sink functions, such as `document.write` or `innerHTML` (HTML sinks) or `eval` (JavaScript sink). Such data can originate from different sources, such as parts of the URL or the referrer (also referred to as *reflected* client-side XSS), from client-side storage such as cookies or LocalStorage (*persistent* client-side XSS), or through PostMessages. Previous work has thoroughly investigated the prevalence of reflected client-side XSS [5, 6, 7], persistent XSS [17], and postMessage-caused XSS [18, 19]. Notably, previous work has only provided anecdotal evidence for improper sanitization [7] or insecure origin checks [18, 19].

2.2. Input Sanitization

Given the aforementioned model of untrustworthy data, which ends up in a dangerous sink, one potential counter-

```
1  // vulnerable, as attacker input is used
   ↪ unfiltered
2  document.write('<img
   ↪ src="https://ad.com/?referrer=' +
   ↪ attacker_controlled + '">');
3
4  // not-vulnerable, as encodeURI encodes " as
   ↪ %22
5  document.write('<img
   ↪ src="https://ad.com/?referrer=' +
   ↪ encodeURI(attacker_controlled) + '">');
6
7  // vulnerable, as encodeURI does not encode '
8  document.write("<img
   ↪ src='https://ad.com/?referrer=" +
   ↪ encodeURI(attacker_controlled) + "'>");
```

Figure 1. Examples of unfiltered flow, correct sanitization, and incorrect sanitization

measure is to deploy sanitization. The basic idea of sanitization is to remove or encode *dangerous* characters before the data hits the sink. The definition of *dangerous* depends on the type of sink (e.g., > and < are problematic in HTML sinks, but not in JavaScript sinks) as well as the exact context in which data is used. Figure 1 shows an example of these intricacies. The code in line 2 is clearly vulnerable, as the attacker has full control over parts of the string written to `document.write` and can simply inject an arbitrary payload, e.g., `"><script>alert(1)</script>`. This allows the attacker to close the `img` tag and add a new script tag with their payload. To fix this issue, the developer might use the built-in function `encodeURI`, as shown in line 5. This function automatically encodes the double quotes and HTML brackets (<>), thereby stopping the attacker from breaking out of the `img` tag. However, looking at line 8, `encodeURI` must not be used when the attacker-controlled data is within a single-quoted attribute. Since the function does not automatically encode single quotes, the attacker can inject `'/onload='alert(1)'/foo='`. This allows them to break out of the attribute, add the event handlers for both successful and failed loading of the image, and consume the remaining single quote to avoid HTML parsing errors. Note that the / is a replacement for the space, which would be automatically encoded by `encodeURI`. However, HTML parsers allow / between attributes of a tag [20] and treat it as whitespace.

This example highlights that sanitization is dependent on both the type of sink as well as the exact context in which attacker-controlled data may be used. And while two code snippets may look almost the same (lines 5 and 8), choosing the same encoding function yields two different outcomes. In a similar vein, developers may misunderstand the details of other APIs, such as JavaScript's `replace` function. This, when called with a string or a regex without the global modifier, only replaces the first occurrence of a character in a string. Therefore, great care must be taken when choosing a sanitizing function for a particular data flow.

Previous research has already investigated such sanitization routines on the server. Balzarotti et al. [21] automatically detected sanitization routines in PHP code and tested their efficacy against a large test suite of XSS payloads collected by the authors. This requires the application under examination to be open source and only detects issues if one of their test cases matches the problem in the routine. It is therefore only able to detect issues that have occurred previously elsewhere. Similarly, Dahse and Holz [22] studied the *correct* usage of sanitization routines on 25 popular PHP applications based on secure data flows. Hooimeijer et al. [23] manually translate sanitizing routines into the BEK language. This representation can then be used to prove properties of the modeled function, such as idempotence or security guarantees. However, their approach relied on the manual translation of sanitizers and therefore does not scale to real-world Web JavaScript. Argyros et al. [24] take a black box approach to infer sanitizers as symbolic finite transducers. In addition, they convert their models into the BEK language to compute equivalence and idempotence checks, and perform an empirical study using 7 server-side applications. The SCRIPTGARD framework [25] uses positive tainting to detect the correctness of sanitizer placement and provide automated runtime sanitization. Their empirical study was performed on a single .NET web application. Weinberger et al. [26] create a web browser model to study sanitization of web frameworks in 8 PHP applications using a combination of manual and automated exploration.

Symbolic analysis of string functions using deterministic finite automata has been studied by Yu et al. [27] and applied to evaluate the correctness of input validation functions (e.g., for E-mail inputs) in client-side JavaScript on 13 websites [28]. Further studies [29] extend this technique to model the effectiveness of sanitizer functions and, where necessary, use differential repair to provide fixes to client and server-side code automatically. Yu et al. [30] use a similar approach to generate sanitizer functions if a vulnerability is discovered automatically. Both of these studies validated their techniques using dynamic slicing to extract sanitization functions from 5 PHP applications.

Orthogonally to academic research, browser vendors and standards bodies have invested in deploying improved client-side filtering mechanisms. On the one hand, the recently proposed Trusted Types standard [31] ensures that data flows cannot enter a sink without passing through a sanitizer function. This way, developers cannot forget to sanitize their data, as sink access with pure strings (instead of trusted types) is disabled. Furthermore, the proposed Sanitizer API [32] aims at providing an easy-to-use interface for developers to sanitize their data. This is motivated by the fact that the most advanced sanitization libraries, such as DOMPurify [33], rely on manual parsing instead of browser built-in DOM parsing. This leaves room for inconsistencies between DOMPurify and actual browser behavior, opening DOMPurify up for possible bypasses. Overall, while DOMPurify provides a strong basis for proper sanitization, developers lack built-in capabilities to sanitize strings, leading them to build their own sanitizers instead.

2.3. OWASP Recommendations

Generally speaking, there are two approaches to sanitization: Firstly, a validator (or filter), will only allow values to pass through if they are deemed safe for the given context. Secondly, a mutating sanitizer will remove or replace potentially harmful fragments of the input. In this work, we focus on mutating sanitizers, as these can be abstracted based on the operations that are conducted on an input. *Secure* sanitizers can either be generic for a sink class (HTML or JavaScript) or specific for the context they are used in. For a generic sanitizer to be secure, it needs to mutate all characters which are dangerous in the sink context. Table 1 highlights such dangerous characters, showing the OWASP recommendation for which characters should be considered dangerous (and hence, must be encoded properly or removed altogether) depending on the injection context.

TABLE 1. CHARACTERS TO BE ENCODED PER SINK CONTEXT

Context	OWASP Recommendation [34]
HTML	`<>'"&` except HTML encoded chars
HTML Attr.	The quote characters (`"` and `'`) as well as characters usable to break out of unquoted attribute values (including: `[space] % * + , - / ; < = > ^` and `|`), properties and event handlers
JavaScript	non-alphanumeric except `,._` whitespace or hex/unicode encoded

When an attacker-controllable piece of data is used within a sink such as `document.write` or `innerHTML`, we refer to this as an HTML context, as these APIs expect HTML to passed to them. Here, we need to distinguish between two cases: 1) the attacker-controlled data is used within a single- or double-quoted attribute, and 2) the data is used outside of such attributes. The latter case is shown in the first line of the table. Here, the OWASP recommendations state that a generic sanitizer should remove >, <, double and single quotes, as well as the `&` (except for when it is used to encoded already encoded characters, e.g., `&lt;` for <). For the former case, the OWASP recommendations go even further to state that any non-alphanumeric characters should be encoded, except those that are used for encoding (e.g., the `&` sign) if used as part of existing HTML entity encoding.
If attacker-controlled data is used within a JavaScript context, e.g., a call to `eval`, the OWASP guide recommends that all non-alphanumeric characters are escaped, with a handful of exceptions such as the comma, period, or underscore.

Note that the OWASP recommendation – especially for the JavaScript context – is very aggressive: it is perfectly possible to construct a secure sanitizer by encoding only a subset of the recommended characters depending on the injection context. This, however, opens the door to the sanitizer being vulnerable if there are minor changes in the surrounding code.

3. Research Questions and Methodology

Based on the explanation in Section 2, it becomes clear that sanitization is very context-specific and difficult to get right. For sinks that can lead to direct code execution (i.e., HTML and JavaScript sinks), there is no standardized sanitization functionality available in the browser. Thus, developers either have to write their own *by hand* (which we refer to as *hand sanitizers*) or use external sanitization libraries (e.g., DOMPurify). The main goal of our work, therefore, is to understand the usage and security of

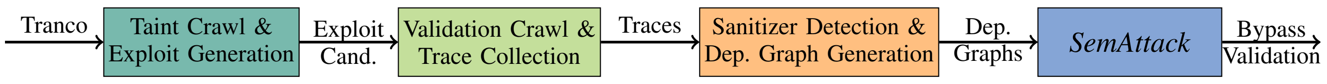
Tranco
Taint Crawl & Exploit Generation
Exploit Cand.
Validation Crawl & Trace Collection
Traces
Sanitizer Detection & Dep. Graph Generation
Dep. Graphs
SemAttack
Bypass Validation

Figure 2. Overview of our sanitizer evaluation methodology.

JavaScript sanitizers on the client-side Web. To this end, we propose two main research questions, which we aim to answer in the following.

RQ1: How many data flows between URL sources and execution sinks use some type of sanitization routine? While prior work has focussed only on finding *exploitable* data flows, we are rather interested in shining light on those cases which could not be directly exploited because of sanitization.

RQ2: What fraction of sanitizers provide sufficient protection against XSS exploits for the specific sink and injection context in which they are found? While understanding the generality of a sanitizer is meaningful to assess its applicability to other data flows, we aim to understand if a chosen sanitizer is sufficient for the exact injection context. For example, while a generic HTML sanitizer needs to encode (at least) <>, it suffices to escape single quotes when the injection is within a single-quoted attribute, thereby ensuring the attacker's payload cannot break out of the attribute.

In order to address these questions, we require a technique to automatically detect, analyze, and validate client-side JavaScript sanitizer functions on a large scale. While prior work has provided anecdotal evidence for the existence of insecure sanitization in client-side Web applications [7], previous studies tend to focus on server-side sanitization [21, 23, 24, 26, 29], rely on manual exploration or translation [23, 24, 26], and perform empirical studies with a small number of hand-picked applications [21, 23, 24, 26, 27, 29]. In fact, Weinberger et al. [26] explicitly exclude client-side XSS sanitization from their studies, stating that "protection for this class of XSS requires further research".

3.1. System Overview

We present our methodology used to answer the above research questions in the rest of this section, as summarized in Figure 2. To start, we collect invocations of dangerous sinks with data originating from URL sources using taint tracking as described in Section 3.2. Based on these taint flows, we generate exploit candidates as described in Section 3.3. On the one hand, this allows us to detect exploitable flaws, but more importantly, it ensures that if sanitizers with conditionals are present, as more of their code paths will be executed. In Section 3.4, we extract traces of the manipulating operations conducted on attacker-controllable data before it entered the respective sinks. Based on this set of operations, we ascertain if the data flow was sanitized, e.g., if the operations are only concatenations, the flow is discarded from further analysis. For the flows that are identified as sanitizers, we then generate dependency graphs, which abstract the functionality of the applied sanitizer. Finally, we pass the graphs to *SemAttack*, our analysis framework, as described in Section 3.5. Here, we evaluate whether or not the sanitizer is secure, i.e., it protects the website against code injection for the given injection context. We achieve this by attempting to find a transformation of the initial exploit payload that defeats the insecure sanitizer. Finally, for each combination of data flow/potentially insecure sanitizer, we build exploit URLs using the targeted payloads especially crafted to circumvent the sanitizer in the given injection context. We then validate these bypasses by visiting the modified URLs to achieve code execution.

3.2. Taint Flow Detection

Taint tracking is a well-established technique for detecting client-side XSS flaws [5, 6]. String data emanating from user-controlled (and therefore also attacker-controlled) *sources* are marked as *tainted* by a modified browser engine. Typical examples of sources are `location.*` and `documentURI` properties. Additional modifications to the browser and JavaScript engines ensure that taint information is correctly propagated during string operations, such as `substr` and `replace`. If a tainted string is detected entering a *sink* function, the path between source and sink is potentially vulnerable to client-side XSS.

The taint browser used for this study (known as project "Foxhound") is based on the open-source Web browser Firefox, version 80, and tracks data flows from sources through both SpiderMonkey, the JavaScript engine, as well as through Gecko, the rendering engine, into sinks. In prior work, which focused on the *detection* of XSS [5, 6, 17], only the taint status of each character (i.e., the source and the usage of built-in encoding functions) is stored. These approaches, however, store insufficient information for the purposes of our study. In order to extract information about any underlying sanitization functions, we require precise knowledge of all string transformations that occur between source and sink. To this end, we follow the method described by Stock et al. [7], whereby the internal string representation is enhanced with a single pointer to a list of taint ranges, with a `null` pointer indicating the string is untainted. Each taint range stores a start and end character, together with a linked list of taint operations we refer to as the *taint flow*. The taint operations include information about sources and sinks, both native and user-defined function calls, together with their arguments and calling context. A tainted string which enters a sink function is denoted a *finding* and each finding consists of one or more taint flows. Figure 3 shows an example of such a finding. The name of the function in which the operation occurred is given in parentheses behind each operation. We will use this finding as a running example throughout the next section. Additional function arguments, as well as location information (i.e., where in the JavaScript source did this call occur), are omitted for brevity if not necessary to illustrate our methodology. The string "LOTSOFCHARS" shown at the bottom enters the `innerHTML` sink. The string consists of static data and tainted data, the latter indicated by shaded boxes in Figure 3. As the two taint flows are part of the same finding, their flows have to join

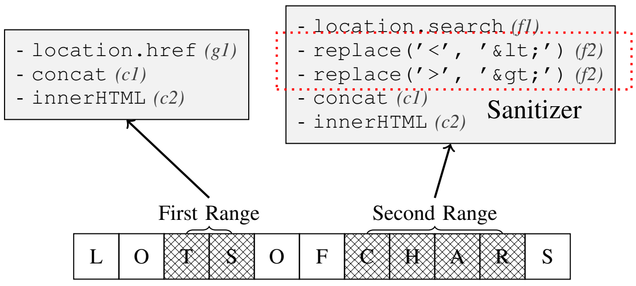

Figure 3. Example of a finding

```
1
2  function sanitize(untrusted) {
3    let re = /[<>]/g;
4    if (re.test(untrusted)) {
5      return untrusted.replace(/</g,
     ↪   '&lt;').replace(/>/g, '&gt;');
6    }
7    return untrusted;
8  }
```

Figure 4. Example sanitizer with conditional statement.

at one point (here, the `concat` call in $c1$ just before the sink).

3.3. Exploit Generation

As discussed by prior work [5], the mere existence of a tainted data flow does not imply an exploitable flaw, which is why such flows require validation through a proof-of-concept exploit. For our paper, the exploitability of a flaw is not as relevant as for prior work, as we focus on sanitizers. However, given our approach of abstracting sanitizer functionality from the applied operations to a tainted string, we aim to trigger as much of the sanitization code as possible. This is motivated by the fact that some operations may only occur if certain conditions are met. Figure 4 shows an example for such a conditional sanitizer. Here, the `replace` operation (line 4) is only conducted if the regular expression (defined in line 2) matches, which occurs when either $<$ or $>$ are in the untrusted input. To achieve this coverage, we, therefore, generate exploit candidates for each detected data flow.

The generation strategies are in line with prior work [5, 7, 17, 35], which is why we only briefly outline them here and refer the reader to these works for a more detailed discussion. In a nutshell, the exploit generator produces an injection-context-specific *breakOut* sequence, a sink-specific *payload*, and an injection-context-specific *breakIn* sequence. As an example, the insecure flow in Figure 1 (line 2) occurs in a double-quoted attribute within an `img` tag. Hence, to break out, we first close the attribute with a double quote, followed by closing of the `img` tag (`">`). Next, we generate a payload to trigger an alert box, e.g., `<script>alert(1)</script>`. In the HTML context, we do not need to generate a specific *breakIn* sequence, as the HTML parser tolerates the `">` suffix (which is hard-coded after the injection point).

In addition, we generate an additional payload for cases where the injection occurs in an attribute context of an HTML element which allows execution of the `onload` and `onerror` event handlers. In this case, we construct a payload by first breaking out of the attribute with a quote (either single or double, depending on the context), setting the event handlers, and finally reentering an attribute. For example, for a double-quoted attribute, the corresponding exploit would be: `" onload=alert(1) onerror=alert(1) foo="`. This construction has the advantage that it will not be blocked by sanitization functions which remove <> characters. We also generate payloads that use template literals (i.e., backticks) to call the reporting function (e.g., `alert`1``) in order to circumvent sanitizers that filter brackets. The generated exploit payloads are thus highly specific to the injection context, i.e., without sanitization they should lead to code execution while also triggering code of sanitizers if they are indeed adequate (or at least attempt to sanitize data).

Based on the exploits generated for the findings derived in the initial crawl, we now conduct a *validation crawl* in which we attempt to validate the generated exploit URLs. Note that this primarily serves to cover more paths in the sanitizer code and not to reproduce findings of prior work.

Note that for our browser engine, we decided to disable automatic encoding of the URL query and fragment. Albeit this behavior is the default in modern browsers and therefore mitigates exploitation of vulnerabilities, our focus is on investigating *intentional* sanitization. Furthermore, legacy browsers like Internet Explorer do not apply such auto-encoding, meaning that developers must not rely on implicit auto-encoding to "secure" their applications, as this would leave legacy clients vulnerable.

3.4. Detection of Hand-Sanitizers

We analyze all findings from the validation crawl in the detection phase as follows: In the first stage, we filter the findings to ensure they originate from attacker-controlled sources such as `location.*` or `documentURI` and flow into sink functions which allow direct code execution, such as `innerHTML` or `document.write`.

In order to extract sanitization functions, we first reconstruct the call graph from the taint flow by determining which operations belong to the same functions and in which order they are called. We then analyze which operations in the call graph perform sanitization. To determine this, we check whether the transformations operate on potentially harmful characters and flag the corresponding operations. Potentially harmful characters are defined as characters that should be encoded according to the OWASP recommendations presented in Section 2.3. For the JavaScript context we limited the characters to syntactically significant characters, such as {, }, `(`, `)`, and so on to avoid flagging most `replace` calls. Generally, these are all the characters that can cause state transitions in the HTML or JavaScript parser, for example, combinations of the $<$, $>$ and / characters can be used to break out of the HTML context. Additionally we flag string operations matching event handlers listed on [36] and built-in functions which encode text values, such as `escape` or `encodeURIComponent`.

For example, a call to `replace(/</g, '&lt;')` has the $<$ character as first argument which should be sanitized for the HTML as well as the HTML attribute context, and thus the replace statement is flagged.

```
var r = data.replace(
  /[<&>]/g,
  function(e) { return "&#" +
    e.charCodeAt(0) + ";" }
);
```

Figure 5. `replace` call with callback argument.

```
var r2 = data.replace(
  /[&<>"'`=\/]/g,
  function(t) { return c[t] }
);
```

Figure 6. Problematic `replace` call with callback argument.

The part of the call graph encompassing all flagged operations is identified as a sanitizer function. It contains all statements aiding the sanitization of the given flow. For each such candidate, we generate an abstract representation of its behavior. Flows without flagged operations, i.e., without sanitization logic, are discarded from further analysis.

The example finding in Figure 3 contains two taint flows. The first contains no operations aiding sanitization and is thus discarded, as it is not of interest for this work. In the second flow, we have two calls to `replace` operating on characters deemed as dangerous according to the OWASP recommendations. These operations both take place in the function $f2$, which is called by the function $f1$ handling the main application logic. We, therefore, detect the function $f2$ to be the sanitizer and prepare it for further analysis as described in the following.

3.4.1. Abstracting the JavaScript Semantics.

The JavaScript language provides a multitude of built-in functions operating on strings which frequently overlap in use. Our analysis, presented in the following, supports only a subset of these functions. The supported functions are: `replace` (With or without the global flag and with both literals and regular expressions as first parameter), `encodeURI`, `encodeURIComponent`, `decodeURI`, `decodeURIComponent`, `JSON.parse`, `JSON.stringify`, `substr`, `trim`, `toUpperCase`, `toLowerCase`, `split`, `escape`, and `unescape`. We first preprocess the extracted sanitizer to only use the aforementioned functions. In the following we highlight some of the preprocessing steps we do in this stage.

Simplifications. `String.substr`, `String.slice` and `String.substring` roughly provide the same functionality. Therefore, we attempt to unify such calls by transforming them to `String.substr` calls, by recalculating the parameter values. Another example are `String.split(v1)` calls directly followed by a call to `Array.join(v2)`. The two operations in combination are equivalent to `String.replace(/v1/g, v2)` thus we transform them into a call to `String.replace`.

Replace with function as second argument. JavaScript allows a function to be a passed as the second parameter to the `String.replace` function [37]. For every match of the search pattern, the function is called, and a dynamic replacement value is calculated. We handle such `String.replace` calls by dynamically evaluating it (the function's source code is available in the taint flow) for all possible input values.

An example for such a call is shown in Figure 5. The function is self-contained, i.e., it does not depend on any objects from an outer scope. We, therefore, can enumerate all values it matches based on the pattern. For each possible match, we call the function to evaluate the replacement value and transform it into its own replace statement. We then model the original replace statement as a sequence of replace statements in the resulting dependency graph. Splitting up a replace statement introduces the possibility for interference. That is, characters resulting from a prior `replace` call are processed again. This is in contrast to a "Regular" replace operation which does a linear scan of the input string and thus avoid such interferences by design. We, therefore, order the operations to avoid any such interferences.
For example for the function in Figure 5 our tool would analyze the regular expression as well as the replace function to yield the following tuples: $[\langle \&, \&\#38; \rangle, \langle <, \&\#60; \rangle, \langle >, \&\#62; \rangle]$. Note that our analysis deliberately puts the `&` character before the $<$ character. In the original order ($<$, `&`, $>$) the result of the replace on $<$ (`&#60;`) would be processed again by the replace operation on `&`. Note that this transformation is only valid for character sets, e.g., `/[&><"']/`, which match a finite number of characters.

This approach also requires the function to be self-contained, such that it does not reference values from outer scopes. A pattern we commonly encountered uses a lookup table from an outer scope to determine the replacement values, as shown in Figure 6. The calculation depends on the Map `c`, which is declared in an outer scope and thus not visible in the taint flow, which only contains the textual representation of the callback function. Therefore, during our analysis, no information about `c` is available. To still be able to analyze such flows, we approximate the result. Such findings are marked as an approximation, and we assume the empty string as the replacement value. This does not cause false positives, according to our evaluation. We validate each sanitizer detected as vulnerable and have not found any occurrences where our analysis flagged a sanitizer as insecure due to this.

3.4.2. Modeling Browser Encoding.

One effective sanitization technique is to use built-in browser functionality such as the `textContent` [38] property of HTML elements. An example of such a sanitizer is given in Figure 7. In this case, a new text element is created with the unsanitized input. On getting the `innerHTML` property, the browser automatically sanitizes the text by HTML encoding the $<$, $>$, `&` and non-breaking space (`0xA0`) characters.

```
function sanitize(untrusted) {
  const trashSpan =
  ↪  document.createElement('span');
  trashSpan.textContent = untrustedText;
  return trashSpan.innerHTML;
}
```

Figure 7. Example sanitizer using `textContent`.

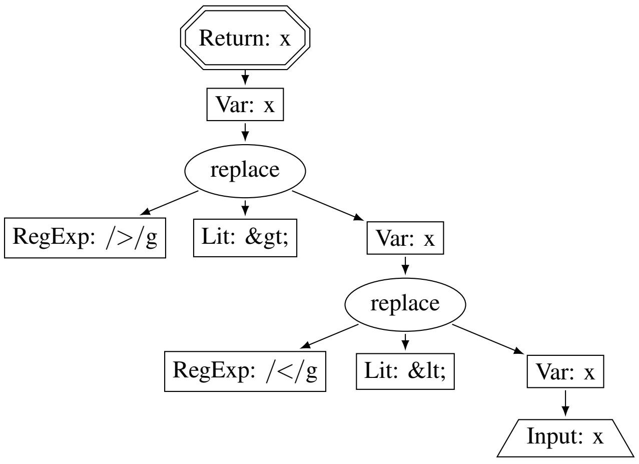

Figure 8. Dependency Graph Example

Our taint-aware browser is able to detect these cases by instrumenting the `EncodeTextFragment` function [39] of Firefox and adding it to the taint flow. In order to evaluate the effectiveness of such sanitizers, we model calls to the `EncodeTextFragment` as a series of replace statements on the characters mentioned above. Similar modelling is performed for the combination of calls to `setAttribute` followed by `outerHTML`. In this case the `EncodeAttrString` function is called which encodes the `"`, `&` and non-breaking space characters, and is modeled in a similar way.

3.4.3. Dependency Graphs. Based on the sanitizer detection and preprocessing described previously, we extracted the slice of the Web application containing the sanitization statements for a given taint flow.

For each such finding/sanitizer combination, we then generate an abstract model describing the data flow of the program slice, known as the *Dependency Graph*, based on the definition of Yu et al. [27]. An example for such a dependency graph (modeling $f2$, the detected sanitizer from the second taint flow in Figure 3) is provided in Figure 8. To correctly identify sanitizers that are identical, we annotate each resulting dependency graph with a *sanitizer hash*. This hash is built over the sequence of transformation operations contained in the dependency graph. Arguments with dynamic values (e.g., `location.href`) are discarded to avoid misclassifications.

In addition to the statements modeling the program's execution, metadata is attached to the dependency graph, containing information such as the domain of the original finding, the execution context of the sink function, and all information required to reconstruct the original exploit. These dependency graphs are then used as the input for the next stage, our automaton-based security analysis.

3.5. Automaton-based Evaluation

In order to evaluate the effectiveness of a sanitization function, we use symbolic string analysis to compute the set of strings allowed at the function's output. If the set of output strings contains values which could lead to XSS, then the sanitizer is labeled as vulnerable.

In our analysis, we use a deterministic finite automaton (DFA) to represent the set of allowed strings after each

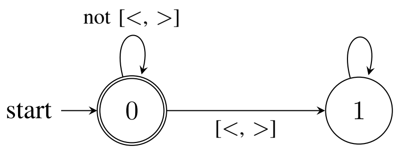

Figure 9. Sample Deterministic Finite Automaton accepting the regular expression `/^[^<>]*$/`.

operation in the dependency graph. A DFA either accepts or rejects an input string by performing a series of state transitions determined by the sequence of characters in the string. A DFA consists of a set of states labeled as either accepting or rejecting. The DFA begins in the initial state, and the next state is determined by the value of the next character in the string. State transitions are performed in sequence until the end of the string is reached. A given input string is accepted if the state machine ends in an accepting state.

We compute the output DFA of a given dependency graph by iteratively computing DFAs for each node in the graph. The DFA of the current node is computed by applying transformations on the previous node's DFA corresponding to the given string operation (e.g., `replace`). We set the input of the dependency graph to the DFA accepting all possible strings values, denoted Σ^*. The DFA obtained at the output of the dependency graph is known as the *post-image*, and represents the set of string which can be returned by the sanitization function. Note that it is possible to compute the same post-image for multiple dependency graphs: for example, any sanitizers comprising only single replace operations will result in a Σ^* post-image.

To illustrate this concept, Figure 9 shows the post-image after applying the dependency graph shown in Figure 8. In this DFA, the initial state is accepting, which means the empty string is accepted. The DFA will remain in the initial state until a $<$ or $>$ character is encountered. In this case, the DFA will transition to state 1, which rejects the string. As there are no transitions out of this state, it is also known as a sink state. As such, the DFA will only accept strings that do not contain <> characters.

We evaluate the effectiveness of a sanitization function by computing the intersection of the post-image with a DFA representing an XSS payload. A non-empty intersection implies that there exists a set of input strings which are transformed by the sanitizer in such a way that leads to an XSS payload at the function's output. We construct the payload DFA as follows: first, we obtain the original taint flow where the sanitization function was discovered from the dependency graph's metadata. The sink function and injection context of the taint flow is then used to generate exploit strings using the method described in Section 3.3. Finally, we compute the payload DFA as the set of strings that contain at least one of the generated exploits. Overall, we define a sanitizer as vulnerable if we discover at least one payload DFA with a non-empty intersection.

For example, consider an instance of the sanitizer shown in Figure 8 with a taint flow into the content of an HTML element via the `document.write` method. In this case we first generate an exploit string of the form `<script>alert(1)</script>`. (*breakOut* and *breakIn* sequences are omitted here for brevity.) The corresponding payload DFA will be `.*<script>alert(1)</script>.*`, that is any string containing the generated payload. The intersection of this DFA with the post-image in Figure 8 will be empty as the < and > characters are removed from the output, and therefore we label the sanitizer as secure. Now consider that we discover a second instance of the same sanitizer with a taint flow into a double-quoted attribute of an image tag. In this case, we generate a payload DFA containing `" onload=alert(1) foo="`. (`onerror` omitted for brevity.) In this context the sanitizer is vulnerable as strings containing the " character are accepted by the post-image DFA. Given that this sanitizer has been found with at least one non-zero intersection, we label it as vulnerable for our analysis.

In order to successfully validate vulnerable sanitizers, we also need to compute the input DFA which corresponds to a vulnerable output. To do this, we perform a second iterative analysis over the dependency graph, but this time starting at the return node and traversing the graph in reverse, applying inverse DFA transformations in turn until the input node is reached. The return node value is set to the intersection of the post-image and the payload DFA, and the resulting DFA at the input node is known as the pre-image. We then generate a single string from the pre-image and use this to construct a modified exploit URL.

Returning to our example, the computed pre-image is simply the set of strings containing `" onload=alert(1) foo="`, as the payload is not transformed by the sanitizer. This will not always be the case, however. Consider a second example with a sanitizer comparing a single replace operation of the form `input.replace('"', '')`, which will replace the first instance of a double-quote character in the string. Assuming the sanitizer is also found in a double-quoted attribute context, the corresponding pre-image will be `"" onload=alert(1) foo="`.

3.5.1. Implementation. Our automaton implementation, referred to as *SemAttack* in the following, is based on *SemRep* [29] and uses the MONA package [40] to represent DFAs as Multi-terminal Binary Decision Diagrams (MBDDs). We made significant enhancements to *SemRep* in order to support the string operations found in modern client-side JavaScript.

One important enhancement was the modelling of operations which only replace the first instance of a search pattern. Examples include string replace operations (e.g., `replace("&", "&amp;")`) and regular expressions which do not use the global flag (e.g., `replace(/[<>]/, '')`). We also implemented function modeling for built-in sanitization operations (e.g., `escape`, `encodeURI`, `JSON.stringify`). With our improvements we were able to model 98.4% of the operations in all examined flows collected in Section 4. In order to successfully model search patterns found in the wild, we enhanced the regular expression engine to include e.g. correct parsing of shorthand classes such as `\d\D\s\S\w\W\p`.

In order to achieve the performance necessary for large-scale sanitizer evaluation, we also enhanced the MONA library to allow thread-safe DFA operations and, therefore, parallel execution on modern multi-core CPUs. In addition, we also added robust error handling and propagation to ensure that runtime exceptions did not cause the entire analysis to crash. During pre-image computation, we observed that some operations could cause the automaton to grow rapidly in size, causing an error as the internal MONA limit on the number of automaton states (2^{24}) is reached. An example of this includes chains of string deletion operations (i.e., `replace(pattern, "")`). In these cases, we create a single example string from the DFA (known as a singleton) and attempt the operation again with the singleton DFA. This DFA represents a subset of the original DFA, and will therefore be smaller and less likely to reach the internal limit of MONA. This approximation is sufficient for our analysis as it is still possible to generate a single exploit URL from the subset. During our empirical study, we were able to successfully generate and validate payloads for sanitizers where this approximation was necessary.

4. Empirical Study

In this section, we apply the techniques described in Section 3 to perform a large-scale analysis of modern client-side JavaScript sanitizer functions.

We conducted our study over 2 weeks between April and May 2021 through a US-based IP. We took the top 20,000 entries in the Tranco [41] list (ID: G4NK) generated on 19th April 2021. We visited each top-level URL, collecting taint flows as described in Section 3.2. In addition, a random sample of 100 links from each top-level URL were extracted and added to the queue of URLs to be visited. We favor a broader crawl with less depth than the work from Lekies et al. [5], yet visit more pages per site than Melicher et al. [6]. We intentionally make this design choice to cover both high number of sites and a broad variety of code on each site.

During our regular crawl we visited 876,872 pages and 4,389,872 frames. The number of taint flows collected from both regular and validation crawls are summarized in Table 2. Out of 124 million findings, we were able to generate 1,746,846 exploit URLs for 3,787 domains. Out of those we could successfully validate 709,683 (40.6%) client-side XSS vulnerabilities.

4.1. Hand-written Sanitizer Study Results

In this section we provide an overview of the sanitizing approaches we encountered during our study. Table 3 summarizes our findings. We discuss some general trends

TABLE 2. CRAWL RESULTS

	Regular	Validation	Total
Findings	124,015,072	55,930,555	179,945,542
Taint Flows	418,342,032	187,097,917	605,439,949
URL → HTML	1,824,752 (†)	19,343,035	21,167,787
URL → JS	172,774 (†)	1,152,973	1,325,747
Examined Flows	0	20,496,008 (*)	20,496,008 (*)

(†) Exploit URLs were generated for these flows
(*) Examined Flows originate in externally controllable sources and flow into a sink allowing script execution.

TABLE 3. SANITIZER ANALYSIS RESULTS

Description	Count	On n Domains
Unique Sanitizer	705	1,415
Post Image	272	1,415
Vulnerable Sanitizer	88 (12.5%)	102

here and give an in-depth exploration of some of the more interesting findings in the following sections.

In the following we will use the taint flows collected in the validation crawl for further analysis. This allows us to detect conditionally executed sanitizer functions such as the one shown in Figure 4. This also removes all domains without findings relevant for client-side XSS from the investigated data set. Thus, we only consider the 20,496,008 flows denoted as examined flows in Table 2 which occurred on 3,787 domains, where *domain* refers to the effective top-level domain plus one (eTLD+1), from here on. The number of domains involved in each step is:

$$20{,}000 \xrightarrow[\text{Flows}]{\text{Dangerous}} 3{,}787 \xrightarrow{\text{with Sanitizer}} 1{,}415 \xrightarrow{\text{Vulnerable}} 102$$

In this section we provide an overview of the sanitizing approaches we encountered during our study. Table 3 summarizes our findings. We discuss some general trends here and give an in-depth exploration of some of the more interesting findings in the following sections.

We discovered sanitizer functions in 9,984,089 taint flows, that is, in roughly half the collected flows. These flows occur on 1,415 domains out of the validation data set of 3,787 domains, where *domain* refers to the effective top-level domain plus one (eTLD+1). The remaining 2,372 domains have directly exploitable taint flows, which we do not consider any further. While the numbers may appear high, they are in line with the findings of prior work, which indicated around 10% of the top 5,000 and top 10,000 sites to be vulnerable [5, 7, 17]. In total we discovered 817 unique sanitizer functions, where uniqueness is determined based on the sanitizer hash as described in Section 3.4.3. Out of these 817 sanitizers we were able to analyze 705. The automaton analysis took just under 30 minutes running on an AMD EPYC 7702P 64-Core processor.

We were unable to analyze the remaining 112 sanitizers due to four different reasons: In 42 cases, parts of the URL were deleted/changed with a call to replace, e.g., `href.replace(location.hash, "")`. This poses an issue for the analysis, as we only observe the *value* of `location.hash`. For the sake of simplicity, assume `location.hash` was initially set to `#foo` in the crawl. To exploit the flaw, *SemAttack* chooses the payload `<script>alert(1)</script>`. Based on the *observed* semantics of the sanitizer, removal of `#foo` has no impact on the payload, though. *SemAttack* is therefore unable to properly model the flow. Due to limitations of our regular expression engine we were unable to successfully parse regular expressions for 5 sanitizers. This is caused by the regular expression engine of our analysis framework not supporting all features implemented by a modern JavaScript engine. The automata of 14 sanitizers were too big to model for the MONA library, which imposes an internal limit on the number of states. This is therefore not a general limitation of the presented approach. 51

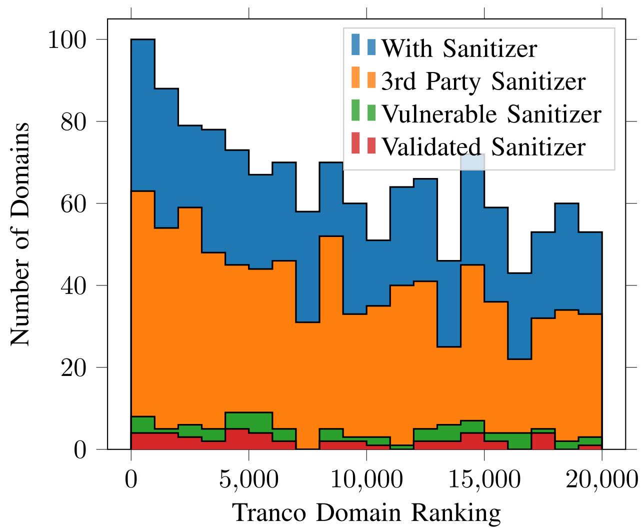

Figure 10. Number of domains with sanitizers ordered by Tranco ranking

sanitizers contained calls to functions not modeled by our framework, such as `DOMParser.parseHTML` or `String.charAt`.

Out of the 705 sanitizers our analysis flagged 88 distributed across 102 domains as insecure for their injection context.

Figure 10 shows the number of domains containing at least one sanitizer and the number of domains containing at least one vulnerable sanitizer, grouped by Tranco list ranking. The graph indicates that highly ranked domains are more likely to deploy a sanitization function than lower ranked sites. For example, sanitizers were found in 100 domains within the top 1,000 ranked domains, compared to 53 in the lowest 1,000 ranked domains considered in our study. Domain ranking appears to have a weaker effect on sanitizer effectiveness, with vulnerable sanitizers found across almost all rankings. The average fraction of domains with a sanitizer which is vulnerable is 7.2%, and remains approximately constant across the domain rankings considered.

4.2. Evaluation

In order to validate our approach, we constructed XSS payloads designed to circumvent the detected sanitizer functions. In this case, we focus on taint flows which are directly exploitable with no additional user interaction, i.e., those flowing from URL-based sources (`location.hash`, `location.href`, `document.documentURI`) into HTML (`document.write`, `document.writeln`, `innerHTML`, `outerHTML` and `insertAdjacentHTML`) or JavaScript (`eval`, `setTimeout` and `new Function`) sinks. For each taint flow, we generated an appropriate XSS payload based on the injection context, following the method described by Lekies et al. [5]. The payload is then converted into a DFA and used as input to the automaton analysis as described in Section 3.

In total we generated 4,093 unique exploit URLs for the 88 sanitizers classified as insecure using the technique described above. Of these, we were unable to validate 10 sanitizers as either the URL was no longer reachable, or the original taint flow could not be found. Of the remaining 78 sanitizers, we were able to successfully trigger JavaScript code execution with at least one URL for 40 sanitizers (51.3%), using a combination of fully-automated (36) techniques and manual inspection (4). Some examples of such bypassed sanitizers are presented in Section 4.3.

Of the 38 unsuccessful exploits, we found the following failure classes: In 20 flows the payload was removed via `String.substring` operations, e.g., by completely deleting the fragment of the URL. This in particular occurred for combinations of `indexOf` and substring operations. In such cases, the JavaScript code would, e.g., determine the index of the first # and cut off the string there. The observed value in our analysis, though, is just a number for which we are unable to ascertain automatically that it is the result of the aforementioned computation. In 6 cases, the exploit payload caused an error in the application's server-side logic. This happens if the query parameter is used to e.g., redirect the user to a specific page. The exploit payload is not a valid value and thus causes an error. In 5 cases we could successfully inject content into a `script.text` sink, but the payload did not lead to a successful exploit due to a non-executable script type (`application/ld+json`). In 4 cases we could successfully identify a sanitization function, but functional code logic outside of the sanitizer prevented code execution. For example, we observed a common pattern whereby parameters are extracted from the URL query using `split('&')` and `split('=')` operations outside of the identified sanitizer. This prevented the successful execution of an event handler exploit which requires the = character. In 2 cases the sanitization depends on heavy usage of branching, that is some functionality is only executed if characters are present in the input string. Thus during validation our transformed payload contains characters not present in the initial payload and thus triggers code paths not seen before. This is an inherent limitation of abstracting the behavior of a sanitizer based on the observed operations. Lastly, in 1 case our regular expression engine did not support parts of the replace pattern (i.e., named groups) and was incorrectly parsed by *SemAttack*.

As shown in Figure 10, successfully validated vulnerable sanitizers appear across the entire range of Tranco rankings considered in this study. The affected sites include several banking sites, popular retailers and businesses, as well as media and news sites.

4.2.1. Ethical Considerations.

Testing XSS payloads on publicly accessible websites comes with a risk of harm to those websites, which we aim to minimize in two ways. Firstly, client-side XSS vulnerabilities are executed in the browser, therefore minimizing the impact on server-side applications. In the large majority of cases, the injected exploit is part of the URL fragment, which is only evaluated in the browser and therefore never reaches the web server. Secondly, our injected payloads call a non-malicious, custom internal logging function, and therefore should not interfere with the behavior of the website.

In addition, successfully validated exploits discovered during this study could be adapted by hackers to perform real attacks. We mitigated this risk twofold, first, we notified all affected website operators before publication, secondly we do not present individual exploit details in this paper or name affected websites.

4.3. Hand Sanitizer Cabinet of Horrors

Based on the results of our automated analysis, we further analyzed the actual code of sanitizers which were prone to be bypassed. In the following, we present several examples of such sanitizers, each of which represents a class of flawed sanitizers discovered in our study.

Regular Expression Limitations. Figure 11 highlights some of the difficulties of trying to sanitize HTML code via regular expressions. The replace statement on line 2 attempts to remove all (opening) script, link or image tags. While the regular expression itself is not problematic, it is not possible to remove all problematic tags this way. As the regular expression will do a linear scan of the string, it is possible to produce e.g., script tags by inserting fragments the replace operation will delete. The following replace statements aim to counteract this issue. The developer, however, failed to take into account that in JavaScript a replace call with a literal as the first argument will only replace the value once.

The payload generated by *SemAttack* which successfully circumvents the sanitizer is given as follows: `#"<><<a>script>alert(1)</script>`. Both *breakOut* and *breakIn* sequences are omitted for clarity in the presented payload. We observe that in order to bypass the replacement in line 2, *SemAttack* injects an `<a>` within the opening `<script>` tag. Hence, the regular expression does not match the script tag, but only the `a` tag and removes it. Afterward, though, the result is a valid opening script tag. Next, the replacements in lines 3 through 6 all only replace the first occurrence. To bypass these operations, we prepend `#"<>` to the payload, which is removed before the sink access in line 7.

```
1  var url = location.href.replace(
2    /<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi,
     ↪  "")
3    .replace('"', "")
4    .replace(">", "")
5    .replace("#", "")
6    .replace("<", "");
7  document.write('<script type="text/javascript"
   ↪  src="example.org?url='+url+'"
   ↪  ></script>');
```

Figure 11. Nested tags pose difficulties for regular expressions

Optimized for Specific Payload. Figure 12 shows a sanitizer which exactly protects against a commonly used payload to demonstrate XSS vulnerabilities, i.e., `alert('xss')`. Notably, though, the sanitizer ignores the ability of JavaScript to rely on Template Strings. These allow to invoke functions even without relying on `()`. Furthermore, the function does not recursively replace the string `alert`. Hence, we can simply modify the payload from `alert(1)` to ``alalertert`1` `` to bypass this filter.

Wrong Context. Figure 13 again highlights how context-sensitive sanitizing statements must be. Notably, the injection occurs within an HTML sink. Hence, the developer

```
function f(v) {
  return v.replace(/'/g, "").replace(/\(/g, "")
  .replace(/\)/g, "").replace(/alert/g, "");
}
```

Figure 12. Sanitizer against specific payload

seemingly built a sanitizer function that encodes `<>` to avoid the injection of a new script tag. However, if we also consider the injection context, we observe that the injection occurs within a double-quoted `src` attribute of an iframe. Here, we do not have to break out of the iframe, but rather add an event listener to it. Specifically, a valid payload (which passes the sanitizer) is `" onload=alert(1) foo=`. This underlines the necessity to not only take the sink into account but also the exact injection context when designing one's own sanitizer.

```
function sanitize(v) {
  return v.replace(/</g, "&lt;")
    .replace(/>/g, "&gt;");
}
var url = 'http://example.org;cat=' +
    sanitize(cat) + '?';
document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Figure 13. Simplified sanitizer for the wrong context

Removing only some Problematic Tags. Figure 14 highlights a lack of understanding what HTML tags can cause code execution. It attempts to filter out `script`, `a` and `img` tags. While these are commonly used to demonstrate XSS vulnerabilities, only removing these tags is insufficient. Simply swapping the `<img>` to the deprecated `<image>` tag suffices to circumvent the sanitizer. Moreover, other elements such as iframes, input fields, or audio also offer support for event handlers which can be abused here. This highlights an additional issue with filtering against a list of problematic tags or attributes. As the Web constantly evolves, HTML elements are added or deprecated frequently. Therefore a blocklisting approach requires frequent updates to stay secure.

```
v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
v = v.replace(
  /<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
  .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
  .replace(/<script(.*)?(\/)?>(.*)?(<\/script>)?/gi, '');
```

Figure 14. Sanitizer matching specific tags

Order of Replace Statements. In addition to the sanitizers observed in our large-scale crawl, we also conducted a prestudy to our work, in which we found one additional interesting case, which we highlight in the following.

Figure 15 shows a sanitizer that would work if the replace statements were swapped. Due to the order of the replace operations, it is trivial to circumvent this sanitizer. The first regular expression attempts to replace opening script tags, followed by the second regular expression, which removes any other tag, such as `<a>`. However, the attacker can still circumvent this to craft a payload that does not contain `<script>` when passing the replace in lines 1 and 2 but does after line 4. Specifically, *SemAttack* produced the following payload: `<<0>script>alert(1)</<0>script>`. The `<0>` tag matches the second replace statement on lines 3 to 4, but its existence prevents the first replace operation (on lines 1 to 2) from sanitizing the input.

```
1  e = e.replace(/[<][s][c][r][i][p][t][^>]*>
2    ([\S\s]*?)<\/[s][c][r][i][p][t][>]/gim, "");
3  e = e.replace(
4    /<\/?\w(?:[^"'>]|"[^"]*"|'[^']*')*>/gim,
   ↪  "");
5  document.write(e);
```

Figure 15. Broken due to statement order

5. Discussion

In this section, we first outline the limitations of our work. Further, we discuss the trends we observed in our large-scale study of sanitization practices on the modern Web. Finally, we identify lessons to be learned from our work that should be taken into account when designing sanitization routines.

5.1. Limitations

Our analysis has certain limitations, which we briefly discuss in the following. We note that obviously, our insights are biased toward high-profile pages and the lack of meaningful interaction (such as login or using existing functionality in the sites) implies that our results are likely a lower bound for sanitization on the Web.

SemAttack contains a regular expression engine that parses regular expressions and turns them into DFAs. While we are able to model most of the encountered expressions, there are several regular expression features we do not support. Those include lazy matching (usually just an optimization), anchors, backreferences and named groups, lookaheads or look behinds. This is an implementation detail and not an inherent limitation of our approach.

Our analysis cannot model taint flows originating from multiple different sources. It is, therefore, unable to generate exploit payloads where data from, e.g., `location.search` and `location.hash` are combined in a way that only by splitting the payload over the two parts of the URL a successful exploit is possible. In this work, we consider each taint flow separately.

If the sanitization functionality is mixed with complex business logic code, our analysis framework fails to terminate due to the automata exploding in complexity. This can happen if the protection code is inlined into the regular code of the application or the collected location information are insufficient to reconstruct the correct call tree and thus includes business logic. This limitation is not general to our approach but purely an implementation detail of the used libraries.

```
v = '<a href="' + elem.url.replace(/"/g,
↪  "&quot;")+ ">";
```

Figure 16. Most specific and minimal sanitizer

TABLE 4. Generality of Analyzed Sanitizing Functions

Context	HTML	HTML Attr.	JavaScript
Unique Sanitizer	169	480	55
Angle Brackets	129 (76.3%)	367 (76.8%)	33 (60.0%)
Double Quote	100 (59.2%)	379 (79.0%)	30 (54.6%)
Single Quote	93 (55.0%)	287 (59.8%)	32 (58.2%)
Backticks	82 (48.5%)	299 (62.3%)	12 (21.8%)
Generic (*)	78 (46.2%)	87 (18.1%)	4 (7.3%)

(*) based on OWASP recommendations (Section 2.3)

5.2. Current State of Sanitization

Our automated way of reasoning about sanitizer semantics allows us to assess the current state of sanitization on the Web, which we present in the following.

5.2.1. Generality. One interesting observation is that most websites deploy sanitizers that are not generic. That is, they only work for the injection context they are used in.

When comparing the set of encoded characters against the OWASP recommendations presented in Section 2.3, most real-world sanitizers encode fewer characters. Table 4 shows which characters sanitizers encode for different contexts. It is interesting to note how few of the sanitizers we encountered conform to the OWASP recommendations. This is possibly rooted in the fact that these recommendations are rather aggressive, and developers instead build more context-specific sanitizers. Notably, as our discussion in Section 4.3 shows, this comes with the increased risk of missing edge cases which in turn allow for bypasses.

An example of a minimal sanitizer is given in Figure 16, which only escapes exactly the `"` character required to break out of the attribute context. Such sanitization routines have the drawback that minimal changes to the surrounding code, e.g., the `href` attribute switching from being enclosed in double quotes to single quotes renders the sanitizer insecure.

Most sanitizers we encountered are neither generic nor minimal. They encode varying amounts of the characters recommended for a given context but rarely all. This is troubling, as these sanitizers seem generic enough to reuse in different places, but they might not prevent XSS in every context.

5.2.2. Usage Patterns. In this section, we investigate the origin of vulnerable sanitizers and their prevalence across the Web. Table 5 shows an overview of the effectiveness

TABLE 5. Comparison of the effectiveness of first- and third-party sanitizers.

Domains	Total	Vulnerable	Validated
Total sanitizer domains	1,415	102 (7.2%)	46 (3.3%)
With first party sanitizer	646	64 (9.9%)	33 (5.1%)
With third party sanitizer	880	41 (4.7%)	15 (1.7%)

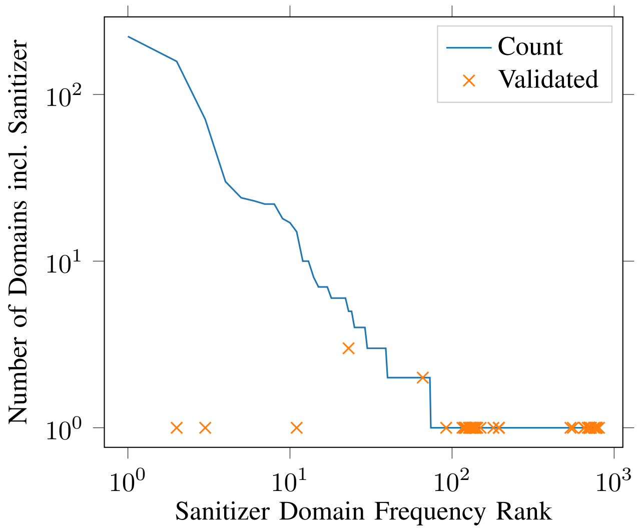

Figure 17. Rank-frequency plot of the number of eTLDs which include a sanitizer from a given domain.

of first- and third-party sanitizers. A third-party sanitizer is defined as a sanitizer function whose script location is hosted on a different domain to the eTLD+1 where it is used. In comparison, a first-party sanitizer is one whose script is hosted on the same domain where it is used. Note that it is possible for a domain to contain both first and third-party sanitizers, such that the number of domains containing first and third-party sanitizers is larger than the total. Table 5 indicates that while third-party sanitizers are more prevalent than first-party sanitizers, vulnerable third-party sanitizers appear on fewer domains.

We also investigated how frequently sanitizers from a particular domain are included on other sites. To do this, we first grouped sanitizers by the domain on which the script containing the sanitizer function is hosted (referred to as the *sanitizer domain* in the following). For each sanitizer domain, we then counted the number of unique domains the sanitizer appeared on. Finally, the list of sanitizer domains was sorted in descending order by the number of domains that included it. The result is the rank-frequency plot shown in Figure 17. The data follow a typical Zipf distribution, with a small number of sanitizers appearing on many domains and many sanitizers included on a few domains. For example, 28.8% of the domains considered include sanitizers from the top three sanitizer domains, whereas 91.0% of sanitizers appear on only a single domain. Sanitizers that are flagged as vulnerable by our analysis are also shown on the plot.

Note that sanitizers from four domains could not be validated as vulnerable for all of the domains on which they were included (represented by the four leftmost crosses in Figure 17). Manual inspection of these cases revealed that scripts containing both vulnerable and non-vulnerable sanitizers were being served from each of the domains.

Overall, vulnerable sanitizers appear on an average of 1.04 domains, compared to 1.93 for all sanitizers. In other words, there are a large number of vulnerable sanitizers which are each used on a small number of domains. These observations support the hypothesis that vulnerable sanitizers are more likely to be written directly by website developers rather than being included from well-tested external libraries.

5.2.3. Standard Sanitizers. Although browsers are starting to experimentally support the Sanitizer API [32], no major browser currently ships with a built-in sanitization routine for HTML or JavaScript. The closest workaround available to developers is to use a combination of `textContent` and `innerHTML` as shown in Figure 7. We found evidence of this behavior in 9.2% of the sanitizers discovered in our study. While this functionality may offer protection against client-side XSS in some contexts, it is not an obvious choice for developers.

A more popular alternative is to use built-in URL encoding functions, such as `encodeURIComponent`, which appeared in 30.8% of sanitizers. Similar functions, such as `encodeURI` and `escape` were found in 1.2% and 6.5% of cases respectively. Besides the fact that `escape` is now deprecated, none of the functions mentioned are sufficient to fully protect against client-side XSS (as demonstrated in Figure 1). Therefore, developers are currently left to rely on third-party libraries or write their own functions.

The third-party library providing sanitization routines we encountered most frequently is the Google closure framework. By matching the generated dependency graphs against the regular expressions used by closure [42] to sanitize input, we were able to detect usage of Closure in 3.2% of examined flows. Notably, this underlines that the vast majority of sanitization on the Web does not occur with widely-used and well-tested libraries but rather that with self-developed and less-tested hand sanitizers.

5.2.4. Upcoming Browser based XSS Mitigations. Browser vendors are aware of these shortcomings and are currently collaborating on working drafts for two XSS mitigation technologies, namely Trusted Types [31] and the aforementioned Sanitizer API [32]. The two proposals are complementary, with Trusted Types aiming to make DOM interaction secure by default via sanitization enforcement, while the Sanitizer API provides built-in sanitizer functionality for HTML contexts. We will detail both proposals in the following.

Trusted Types changes how developers interact with XSS sinks so that they accept trusted values as arguments instead of raw Strings. These trusted values, e.g., TrustedHTML for HTML sinks, must be created by calling a so-called policy, which is registered earlier in the program by the developer. These policies effectively define sanitizing functions for three different contexts: HTML, JavaScript and script URLs. An example on how such a policy is defined and used is provided in Figure 18. Trusted Type enforcement for XSS relevant sinks is enforced via options in the CSP. For the example, setting a CSP such as `require-trusted-types-for 'script';` causes the unsafe assignment on line 6 to throw an type error as the `innerHTML` sink requires `TrustedHTML`.

The security of the sanitizers present in the policies is explicitly left to the developer. Therefore the Trusted Types proposal does not mitigate the risk of broken hand sanitizers. It would be perfectly possible, for example, to include one of the anti-patterns described in Section 4.3 as the sanitizer function in line 2 of Figure 18. In fact, despite the 2021 report into the state of Trusted Types [43] stating that "more than half of the DOM XSS root causes were due to bugs in HTML sanitizers" this proposal explicitly does not attempt to solve this issue.

```
1 const p = '<img src=x onerror=alert(1)>';
2 htmlPolicy = trustedTypes.createPolicy('sanitize', {
3   createHTML: s => s.replace(/\</g, '&lt;')
4 });
5 node.innerHTML = htmlPolicy.createHTML(p);
6 node.innerHTML = p; // unsafe
```

Figure 18. Creating and Using a Trusted Types Policy

```
let sanitizer = new Sanitizer();
let payload = '<img src=x onerror=alert(1)>';
node.setHTML(payload, sanitizer);
let sanitized = sanitizer.sanitizeFor('div', payload);
node.replaceChildren(...sanitized.childNodes);
```

Figure 19. Usage of the Sanitizer API

Nevertheless, by making modern web frameworks such as Angular compatible with Trusted Types, a significant number of websites can gain XSS protection with no changes required to user code [44].

The Sanitizer API, on the other hand, adds sanitizer functionality for HTML contexts to the standard JavaScript environment. As it is built into the browser, the Sanitizer API can reuse the browser's HTML parser machinery and thus eliminate all issues stemming from diverging behavior between the parsing functionality available to developers and how the browser actually interprets HTML. However, due to the context sensitivity of sanitization, the Sanitizer API requires its users to be very explicit about the context in which the output will be used. This requires more code changes by developers and is therefore more difficult to use as a drop-in replacement. An example of the API's usage is shown in Figure 19 and results in a node containing `<img src=x>`.

Unlike the sanitizers we considered during our study, both the Sanitizer API as well as the Trusted Types machinery do not perform string-to-string transformations. Instead they return typed objects encapsulating the sanitized input. This makes it impossible to directly mutate the sanitized value via string operations – a common coding pattern according to our study. This prevents changes to the sanitized value which might alter how the string is parsed, potentially reintroducing XSS vulnerabilities.

Returning to the (in)security of sanitizing functions, Trusted Types still allow for broken sanitizers to be registered as policies. The Sanitizer API on the other hand aims to eliminate broken sanitizer usage by making a secure alternative easily available to web developers. The combination of both, enforcement and a secure sanitizer, would make for an universal XSS mitigation. However it is currently not possible to combine both approaches in an always secure fashion.

5.3. Key Insights

Our analysis has shown that sanitization on the client-side Web is brittle and highly specific to the injection context. In particular, with respect to HTML, there are no built-in functions that allow parsing and sanitization of HTML. Hence, as observed in Section 4.3, developers use methods that are unfit, such as blocklisting certain keywords (such as `alert`) or relying on regular expressions to parse HTML. This is not only infeasible given that HTML is a context-free language, which can therefore not be represented through regular expressions in its entirety. Second, browsers are error-tolerant, leading to attack classes such as mutation-based XSS [45], rendering regular expression parsing dangerous [10].

For JavaScript sinks, developers also often rely on built-in functionality that actually serves different purposes (namely URL encoding). We found several instances where developers relied on built-in functions which are unfit for the purpose (i.e., `encodeURIComponent`, `escape` and `encodeURI`) depending on the surrounding context. And even in cases where this "sanitization" was sufficient, subtle changes to the surrounding code, e.g., swapping double-quoted attributes to single quotes, could render the "protection" useless. Despite their shortcomings, these operations are frequently used.

Moreover, developers seem to misunderstand the intricacies of certain constructs, most prominently the `replace` functionality. The behavior of the `replace` operation differs between three cases: 1) a string literal is used as the needle, 2) a regular expression is used as the needle, and 3) a regular expression with the global flag is used. Contrary to other programming languages (such as Python, PHP, or Java), if invoked with a string, the default behavior of `replace` is to only replace the first occurrence of the pattern. The same applies to the case with a regular expression (without the global flag). This leads to sanitization attempts such as the one shown in Figure 11.

Finally, attempts at complex solutions are often destined to fail. Generally speaking, it is possible to write a secure sanitizer using regular expressions (as is done in the Closure compiler). Their promising approach is not to attempt to parse the structure of the input at all. Purely encoding the characters described in Table 1 is sufficient to write a secure sanitizer. This may, in turn, have an impact on functionality (e.g., because parts of a URL are encoded and the server misunderstands them), but such encoding is secure.

5.4. Calls to Action

Our results highlight the fact that developers are often forced to rely on unsuitable constructs for sanitization and regularly lack knowledge about the intricacies of JavaScript (such as the `replace` behavior) and the specifics of potential XSS payloads (e.g., forgetting to remove `iframe` tags in a sanitizer). All of these aspects highlight the need for browsers to include support for input sanitization. Such built-in support would also benefit from automatic updates, as even for a well-maintained project like DOMPurify, new bypasses are found regularly. Furthermore, ECMA should consider updating the specification for built-in functions such as `replace` to align them with other programming languages developers might be familiar with, e.g., by changing the semantics of `replace` to `replaceAll` and making `replaceOnce` explicit.

6. Conclusion

In this paper, we studied the prevalence and security properties of sanitization routines which aim to protect against client-side XSS in the wild. To this end, we first built a crawling framework to collect taint flows and operation traces during page execution. Based on these traces, we then automatically classified certain flows as having passed through a sanitizer and extracted the operation slices from the taint flow for further analysis. To automatically reason about the (in)security of sanitizers, we then developed *SemAttack*, an automaton-based approach which is able to determine the complete set of outputs a given sanitizer can produce. If a potentially dangerous output is detected, *SemAttack* was able to automatically transform the exploit payload such that the actual payload survives sanitization attempts.

Using these techniques we detected 705 different sanitization routines on 1,415 domains out of the top 20,000 most popular websites. Our analysis classified 88 sanitizers as insecure for the injection context they were used in, and in 40 cases we were able to generate sanitizer-bypassing exploit payloads which successfully triggered JavaScript execution. We found that vulnerable sanitizers are present across the entire range of website rankings considered in the study and that sanitizers written by website developers are more likely to be vulnerable than those included from third-party domains.

Our findings highlight the lack of intuitive and appropriate tools available to JavaScript developers to write generic and secure sanitizers. This confirms the urgent need for a standardized sanitization API available directly in the browser. We thus encourage browser vendors to adopt the currently proposed draft for such an API [32].

Acknowledgments

We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. This work has also received funding from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206.

References

[1] OWASP Foundation Inc, "OWASP Top 10 – 2013 – The Ten Most Critical Web Application Security Risks," https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf, 2013, accessed: 16.09.2021.

[2] ——, "OWASP Top 10 – 2017 – The Ten Most Critical Web Application Security Risks," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2017, accessed: 23.07.2021.

[3] ——, "OWASP Top 10 – 2021," https://owasp.org/Top10/, 2021, accessed: 16.09.2021.

[4] A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," *Web Application Security Consortium, Articles*, 2005.

[5] S. Lekies, B. Stock, and M. Johns, "25 Million Flows Later: Large-scale Detection of DOM-based XSS." in *ACM CCS*, 2013.

[6] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting." in *NDSS*, 2018.

[7] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, “From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting.” in *ACM CCS*, 2015.

[8] T. Nidecki, “Mutation XSS in Google Search,” https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search/, 2019, accessed: 16.09.2021.

[9] V. Kumar, “$20000 Facebook DOM XSS,” https://vinothkumar.me/20000-facebook-dom-xss/, 2020, accessed: 16.09.2021.

[10] D. Bates, A. Barth, and C. Jackson, “Regular Expressions Considered Harmful in Client-Side XSS Filters,” in *WWW*, 2010.

[11] “RegEx match open tags except XHTML self-contained tags,” https://stackoverflow.com/a/1732454, accessed 09.04.2021.

[12] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, “CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy,” in *ACM CCS*, 2016.

[13] M. Weissbacher, T. Lauinger, and W. Robertson, “Why is CSP failing? Trends and challenges in CSP adoption,” in *RAID*, 2014.

[14] S. Calzavara, A. Rabitti, and M. Bugliesi, “Content security problems?: Evaluating the effectiveness of content security policy in the wild,” in *ACM CCS*, 2016.

[15] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, “Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies,” in *NDSS*, 2020.

[16] Cert/CC, “CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests.” https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=496186, Februar 2000, accessed 09.04.2021.

[17] 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 *NDSS*, 2019.

[18] S. Son and V. Shmatikov, “The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites.” in *NDSS*, 2013.

[19] M. Steffens and B. Stock, “PMForce: Systematically Analyzing postMessage Handlers at Scale.” in *ACM CCS*, 2020.

[20] WHATWG, “HTML Living Standard,” https://html.spec.whatwg.org/multipage/parsing.html#parse-error-unexpected-solidus-in-tag, April 2021, accessed 09.04.2021.

[21] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications.” in *IEEE Symposium on Security and Privacy*, 2008.

[22] J. Dahse and T. Holz, “Experience Report: An Empirical Study of PHP Security Mechanism Usage,” in *International Symposium on Software Testing and Analysis*, 2015.

[23] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, “Fast and Precise Sanitizer Analysis with BEK.” in *USENIX Security Symposium*, 2011.

[24] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, “Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters,” in *IEEE Symposium on Security and Privacy*, 2016.

[25] P. Saxena, D. Molnar, and B. Livshits, “SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications,” in *ACM CCS*, 2011.

[26] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. Shin, and D. Song, “A Systematic Analysis of XSS Sanitization in Web Application Frameworks,” in *ESORICS*, 2011.

[27] F. Yu, M. Alkhalaf, and T. Bultan, “Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses,” UC Santa Barbara, 2009-11, Tech. Rep., June 2009.

[28] M. Alkhalaf, T. Bultan, and J. L. Gallegos, “Verifying Client-Side Input Validation Functions using String Analysis,” in *International Conference on Software Engineering*, 2012.

[29] M. Alkhalaf, A. Aydin, and T. Bultan, “Semantic Differential Repair for Input Validation and Sanitization,” in *International Symposium on Software Testing and Analysis*, 2014.

[30] F. Yu, M. Alkhalaf, and T. Bultan, “Patching Vulnerabilities with Sanitization Synthesis.” in *International Conference on Software Engineering*, 2011.

[31] W. I. C. Group, “Explainer: Trusted Types for DOM Manipulation,” https://github.com/WICG/trusted-types, October 2017, accessed 09.04.2021.

[32] Web Incubator Community Group, “HTML Sanitizer API,” https://github.com/WICG/sanitizer-api, September 2020, accessed 09.04.2021.

[33] M. Heiderich, C. Späth, and J. Schwenk, “DOMPurify: Client-Side Protection against XSS and Markup Injection,” in *ESORICS*, 2017.

[34] OWASP Foundation Inc, “Cross Site Scripting Prevention Cheat Sheet,” September 2020, accessed 23.07.2021.

[35] S. Bensalim, D. Klein, T. Barber, and M. Johns, “Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis,” in *Proceedings of the 14th European Workshop on Systems Security*. ACM, 2021.

[36] OWASP Foundation Inc, “XSS Filter Evasion Cheat Sheet,” https://owasp.org/www-community/xss-filter-evasion-cheatsheet, September 2020, accessed 23.07.2021.

[37] MDN contributors, “String.prototype.replace(),” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace#Specifying_a_function_as_a_parameter, July 2020, accessed 09.04.2021.

[38] “DOM Living Standard,” https://dom.spec.whatwg.org/#dom-node-textcontent, accessed 22.07.2021.

[39] “Mozilla Central Mercurial Repository,” https://hg.mozilla.org/mozilla-central/file/default/dom/base/nsContentUtils.cpp, accessed 22.07.2021.

[40] N. Klarlund and A. Møller, *MONA Version 1.4 User Manual*, BRICS, Department of Computer Science, University of Aarhus, January 2001, notes Series NS-01-1. Available from `http://www.brics.dk/mona/`.

[41] V. Le Pochat, T. van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, “Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation.” in *NDSS*, 2019.

[42] Google Inc., “Google Closure Templates,” https://github.com/google/closure-templates/blob/master/javascript/soyutils_usegoog.js#L2480, accessed 05.06.2021.

[43] K. Kotowicz, “Trusted types - mid 2021 report,” https://research.google/pubs/pub50512/, Google Research, Tech. Rep., 2021.

[44] P. Wang, B. Á. Guðmundsson, and K. Kotowicz, “Adopting Trusted Types in Production Web Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study,” in *IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2021.

[45] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, “mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations,” in *ACM CCS*, 2013.