Combinatorially XSSing Web Application Firewalls

Bernhard Garn
SBA Research
Vienna, Austria
bgarn@sba-research.org

Daniel Sebastian Lang
Vienna University of Technology
Vienna, Austria
e1005115@student.tuwien.ac.at

Manuel Leithner
SBA Research
Vienna, Austria
mleithner@sba-research.org

D. Richard Kuhn
NIST
Gaithersburg, MD, USA
kuhn@nist.gov

Raghu Kacker
NIST
Gaithersburg, MD, USA
raghu.kacker@nist.gov

Dimitris E. Simos
SBA Research
Vienna, Austria
dsimos@sba-research.org

***Abstract*—Cross-Site scripting (XSS) is a common class of vulnerabilities in the domain of web applications. As it remains prevalent despite continued efforts by practitioners and researchers, site operators often seek to protect their assets using web application firewalls (WAFs). These systems employ filtering mechanisms to intercept and reject requests that may be suitable to exploit XSS flaws and related vulnerabilities such as SQL injections. However, they generally do not offer complete protection and can often be bypassed using specifically crafted exploits. In this work, we evaluate the effectiveness of WAFs to detect XSS exploits. We develop an attack grammar and use a combinatorial testing approach to generate attack vectors. We compare our vectors with conventional counterparts and their ability to bypass different WAFs. Our results show that the vectors generated with combinatorial testing perform equal or better in almost all cases. They further confirm that most of the rule sets evaluated in this work can be bypassed by at least one of these crafted inputs.**

***Index Terms*—combinatorial testing, security testing, web application, xss, web application firewall**

I. Introduction

Cross-Site Scripting (XSS) is a form of injection attack where an adversary supplies malicious input to a web application that is later transmitted to and executed in the context of the victim's browser (commonly as JavaScript code). Typically, this type of attack is possible when an application requires user input, but does not validate or sanitize this data to make sure that only safe input is processed.

Despite sustained attention from both academic and industrial researchers and consistent visibility amongst developers through resources such as the Open Web Application Security Project's (OWASP) list of *Top 10 Web Application Security Risks* [48], XSS flaws remain a widely prevalent and critical security issue for web applications.

SBA Research (SBA-K1) is a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

Moreover, this work was performed partly under the following financial assistance award 70NANB18H207 from U.S. Department of Commerce, National Institute of Standards and Technology.

To protect sites against exploitation of these vulnerabilities, operators often rely on web application firewalls (WAFs) [20]. These security systems operate on HTTP traffic and filter malicious requests before they can reach the actual application. However, studies on the effectiveness of WAFs show that they do not provide perfect protection from SQL injection attacks [1]. It is not yet clear to what degree a WAF can prevent XSS flaws from being exploited.

A large body of research deals with different aspects of XSS attacks and defensive mechanisms [20]. A variety of software testing methods have been used in order to improve XSS vulnerability detection, including search-based testing [5], unit testing [26], mutation-based testing [24], [39] as well as evolutionary approaches [12], [13], [45].

Web vulnerability scanners provide an automated way to identify XSS flaws and other security issues. However, research has shown that these products have low detection capabilities and a high rate of false positives (in other words, they often classify unsuccessful attacks as successful). Moreover, these tools cannot guarantee a certain coverage of the input space, making it difficult to estimate the reliability of the results [17]. While many works in the literature were successful in exploiting XSS vulnerabilities in web applications, few consider the presence of additional security mechanisms such as WAFs.

In this paper, we propose an attack model for the combinatorial derivation of XSS attack vectors with the goal of *bypassing* or *evading* the filters imposed by WAFs. From a software testing perspective, we consider a WAF as the *system under test* (SUT). We do not develop specific exploits against the web applications protected by these systems. In a case study, we evaluate the performance of the XSS attack vectors generated using the method described in this work. We further compare them against three traditional state-of-the-art lists of XSS attack vectors. Our results show that our approach performs as well or better against almost all SUTs considered herein.

Combinatorially instantiated attack grammars have previously been used for the creation of XSS attack vectors targeting web applications [8], [9], [19], [41]. In contrast, the

method presented in this work evaluates the ability of WAFs to correctly classify malicious requests, thus exploring a novel application domain of combinatorial security testing [40].

Contribution. In particular, this paper makes the following contributions:

- An attack grammar modelling XSS attack vectors for bypassing WAFs,
- Evaluation of combinatorial test sets of different strengths based upon the proposed grammar in a case study against seven SUTs,
- Performance comparison between results achieved by combinatorial test sets of XSS attack vectors and traditional static state-of-the-art lists of attack vectors.

This paper is structured as follows. In Section II, we introduce basic concepts used throughout this work. Section III explains the process and components of our proposed testing approach as well as the developed attack model. In Section IV, we introduce the chosen SUTs, our experimental setup, as well as the selection of static lists of attack vectors for comparison. The following Section V presents and evaluates the results of the case study. Section VI gives an overview of related work, while Section VII contains a discussion of potential threats to validity. Finally, Section VIII concludes this work and provides an outlook for future research.

II. Preliminaries

This section gives an overview of the basic concepts used in this work. In Section II-A, we describe web application firewalls and how they are used to protect web applications from attacks. Finally, we outline *penetration testing* and *combinatorial security testing* for XSS vulnerabilities in Section II-B.

A. Web Application Firewalls

Web application firewalls are conceptually similar to traditional firewalls, offering capabilities to filter traffic based on user-defined rules. However, instead of inspecting network traffic on OSI layer 4 and below [11], they are primarily focused on application layer traffic, particularly HTTP requests. They are often implemented as web server modules or dedicated appliances that are logically placed in front of a web server. When a WAF detects malicious input, it blocks the request and will not forward it to the application. This provides an additional layer of protection by preventing the exploitation of both known and unknown issues in vulnerable software.

When verifying the correctness of filters, it is often important to consider the complexity of the rules employed to make a decision on whether to forward, drop or otherwise process some unit of traffic. Most rules used by traditional firewalls, particularly those operating on lower layers, inspect fields with bounded values: There is a finite number of IP addresses, network protocols, ports and so on. Generating test cases for these rules is relatively simple and could potentially even be performed exhaustively (i.e. by iterating over all possible values of all inspected fields). In contrast, WAFs operate on almost entirely unconstrained user input (i.e. strings), which yields a much larger search space that would have to be considered in order to identify all possible attacks. Furthermore, deciding whether a given user input is actually malicious or not may require additional information. The string `<script>alert(1);</script>` is a classic example for an XSS attack. However, in a message board about programming or security topics, this string might also in fact be a valid content for a comment that *describes* XSS. Without additional specifications, a WAF will have to make certain assumptions about the semantics of the software it is meant to protect. A stricter configuration generally leads to a greater amount of requests that will be blocked, at the risk of increasing the chances of legitimate requests being rejected. Therefore, a balance has to be found between maximum security and little to no false positives (i.e. benign requests that were blocked in error).

A request addressed to an application first reaches the WAF. If it is deemed benign, it is forwarded to the application and processed. The corresponding response is then returned and passed through to the user. This process can be seen in Figure 1. In contrast, Figure 2 visualizes the flow of network traffic that is deemed malicious. Instead of being forwarded to the application, the WAF immediately returns a response to the user, commonly indicating that the submitted request has been denied. In this case, the (possibly malicious) request never reaches the intended target and is thus incapable of achieving exploitation.

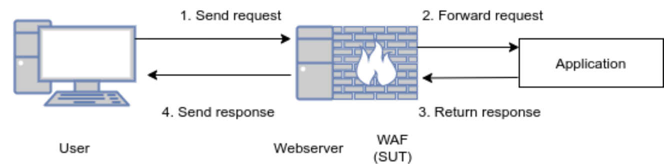

Fig. 1: Request process with benign content or undetected attack.

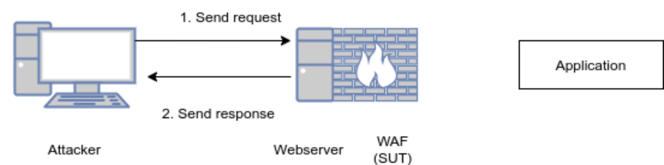

Fig. 2: Request process with detected malicious content.

B. Combinatorial Security Testing

a) Software security testing: Software security testing, an integral part of a properly implemented *software development life cycle* (SDLC) [10], [21], [25], [27], [47], [15], [33], seeks to find vulnerabilities in a given piece of software, which is usually referred to as the SUT.

b) Penetration testing: While security testing as part of an SDLC is often performed in a *white-box* setting, i.e. one where the tester has full access to the source code and internal

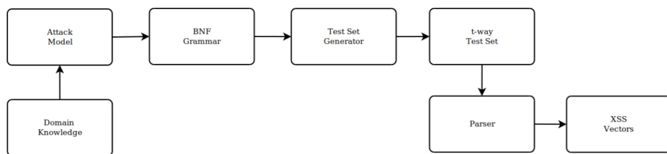

Fig. 3: Modeling process and test generation; taken from [41].

documentation of an application, *penetration testing* seeks to emulate the view of an attacker and is thus usually performed in a *black-box* setting. Additionally, penetration testing is not restricted to identifying vulnerabilities, but also encompasses an array of other activities, from the initial reconnaissance and enumeration phases to the final reporting stage. It may include a larger scope than pure software (security) testing, focusing on the overall risk to the business. [4], [34].

c) Combinatorial security testing: Combinatorial testing is a model-based approach: the SUT is described with finitely many parameters, each of which can take finitely many values. This abstract model of the SUT is termed an *input parameter model* (IPM), and parameter values for the model are commonly provided by application domain experts. When security domain knowledge is used to create an IPM modeling security properties of a system or attacks on a system to be used with CT, it is called *combinatorial security testing* (CST). This approach has been successfully applied to several different kinds of attacks in the general domain of information security [40]. The individual test cases in these generated t-way test sets constitute *abstract attack vectors*, which will subsequently be translated into a domain-specific representation to be executed against a SUT. Note that we follow the terminology used in [8] when referring to test cases as attack vectors.

To make the overall process more tangible, we illustrate the CST approach used in this work in the domain of testing for XSS vulnerabilities with an example. Figure 3 depicts the individual phases involved.

We start with a context free grammar for a test case <test> in BNF:

```
<test>  ::= <tag> <space> <quote>
<tag>   ::= script | img | a
<space> ::= ␣ | \n | \r \ | \0 | <empty>
<quote> ::= ' | " | ` | <empty>
```

Strings derived from this grammar are to be interpreted as a basic and preliminary form of XSS attack vectors. Note that for testing purposes, there are many ways to sample strings in the language that this grammar generates, i.e. there are different *strategies* on how to select test cases. In this work, we employ a combinatorial sampling strategy. Specifically, this grammar can used to define an IPM with three parameters, which are given by the three non-terminal symbols <tag>, <space> and quote. The selected values of these parameters are then concatenated to form a single test case <test>. The elements of this test case may additionally undergo a *translation* step in order to transform it from an abstract test case (i.e. row in a t-way test set) to a *translated* (i.e. ready-to-be executed) test case. We call a function that performs this kind of concatenation and transformation, as well as optional mutations such as changing the case or encoding of strings, a PAYLOAD GENERATOR. In effect, the generation of XSS attack vectors consists of two steps: The first is the generation of a t-way test set based on the IPM, while the second is the construction of translated test cases by the PAYLOAD GENERATOR function. An example for a translated test case (and thus a XSS attack vector) derived from this IPM would be `script\r"`.

Note that based on the IPM specified by the grammar above, we could generate $3 \times 5 \times 4 = 60$ different test cases in total, which would correspond to an exhaustive test set. A pairwise test set, however, exists with only 20 test cases. This pairwise test set achieved a reduction of about 60% in terms of test set size compared to the full input space defined by this IPM.

III. Methodology and Attack Model

This section describes the core of our approach for testing for XSS attack vectors capable of bypassing WAFs. We present an overview of our methodology in Section III-A and discuss our developed attack model in detail in Section III-B. The description of the used testing oracle in this work in given in Section III-C, where we also explain the advantages and drawbacks of choosing the WAFs themselves as test oracles.

A. Testing process overview

Figure 3 shows an overview of the security testing methodology followed in this work, the phases of which are discussed below. Note the WAFs themselves appear twice in our testing methodology. First, they appear as SUTs, when we speak of them as *evaluation targets*. Second, they are implicitly used as oracles when their blocking decisions are used to determine whether a request was actually able to bypass them.

In order to facilitate the automated execution of our approach, we implemented a prototype Python tool that performs the test set generation and translation, execution of test cases (i.e. the submission of XSS attack vectors), and logging of results to a PostgreSQL relational database.

a) Testing environment: We use Docker[1] and Docker-compose[2] to run each SUT in an isolated container with its respective configuration. Figure 4 shows the logical network layout of the virtualized test environment. All SUTs run in their own container and are isolated from the attacking client (i.e. our prototype tool). This not only ensures that no unwanted interactions between the client and the SUTs take place, but also makes adding new SUTs very easy, since they are not tasked with additional steps such as saving evaluation results and can thus remain unmodified.

[1]https://www.docker.com
[2]https://docs.docker.com/compose/

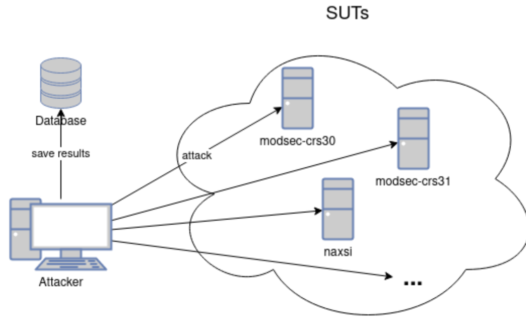

Fig. 4: Network topology.

b) Attack vector generation phase: In the attack vector generation phase, the test sets consisting of XSS attack vectors are generated. For the CST approach to XSS attack vector generation presented in this work we employ several t-way test sets of varying strength. The underlying attack grammar (i.e., IPM) will be described in Section III-B, while the created test sets are described in Section IV-C.

For the purpose of comparing the results achieved by our attack vectors with existing approaches, we also include three lists of XSS attack vectors. Details of these lists are given in Section IV-B.

c) Attack phase: Before this phase begins, the containerized SUTs that were specified in the preparation phase are launched. For one or more specified SUTs, our tool first loads the respective set of XSS attack vectors from the database and then submits them using the *requests* library for Python3 [35]. In most cases, the payload is transmitted as the value of a HTTP GET parameter, since these parameters are scanned in all rules by every tested WAF in this work. This behavior had to be adjusted in some cases; see Table II and Section IV for details. The information on whether the SUT blocks the request or passes it on is saved in our database. We provide more details on the oracle in Section III-C. Note that we ignore what happens to the request after it passes the WAF, since the decision of the oracle is already known at this point.

d) Evaluation phase: After all experiments have been performed, the results of the attack phase can be analyzed and evaluated. Details on this step are presented in Section V.

B. Attack Model

The attack grammar in the form of an IPM devised for testing WAFs for XSS vulnerabilities contains 12 parameters and is given below. The number of distinct values of the respective parameter is enclosed in parentheses.

```
FOBRACKET (1): '<'
TAG (6): 'img', 'script', 'body', ...
FCBRACKET' (1): '>'
QUOTE (4): '"', "'", ...
SPACE (9): "\n", "\t", "\r", ...
EVENT (11): 'data', 'href', 'src', ...
TAG_CASE (3): 'lower', 'upper', 'mixed'
EVENT_CASE (3): 'lower', 'upper', 'mixed'
PAYLOAD (8): '''confirm(1)''',
             '''javascript:alert(1)''',
             '''alert`1`''', ...
LOBRACKET (1): '</'
LCBRACKET (1): '>'
IS_DOUBLE_ENCODED (2): "True", "False"
```

Note that this IPM contains so-called *meta-parameters* for XSS attack vector generation which, in contrast to the parameters given in the example IPM in Section II, encode specific *translation options* for the PAYLOAD GENERATOR function and do not appear in the final XSS attack vectors.

In particular, the meta-parameters in the IPM above describe and control mutations that are commonly applied to attack vectors in security testing with the goal of evading or bypassing filters employed by security software like WAFs. The meta-parameters "TAG_CASE" and "EVENT_CASE" control the casing for the values of the "TAG" and "EVENT" parameters, respectively, which can take the values of "lower", "UPPER" or "mIxEd". Another meta-parameter, "IS_DOUBLE_ENCODED", determines whether to apply URL encoding during generation. As every XSS attack vector is automatically encoded at request time by the Python3 *requests* library, setting this meta-parameter to "True" will result in the vector being URL encoded twice in total.

A simplified version of the PAYLOAD GENERATOR function used in this work is given in Algorithm 1.

Algorithm 1: A PAYLOAD GENERATOR with its execution flow controlled by a meta-parameter.

```
1 Function payload_generator(tag, payload,
    double_encoding) : string is
2     item ← "<" + tag + ">" + payload + "< /" + tag
        + ">";
3     if double_encoding then
4         item ← urlencode(item);
5     end
6     return item;
7 end
```

C. Oracle

The oracle is the component that decides if a test case constitutes a successful attack. Depending on whether this decision was correct, four different outcomes are possible:

	Oracle True	Oracle False
Actual True	True Positive	False Negative
Actual False	False Positive	True Negative

TABLE I: Classification outcomes for attack vectors.

Ideally, a WAF would produce no false negatives or false positives. In our study this means that a WAF should block all attacks (a situation that corresponds to a true positive), but pass all benign requests (thus exhibiting a true negative). Compared to similar case studies on SQL injections [1], [2],

it is more difficult to determine if an XSS payload actually exploits a vulnerability, because the injected code is executed in the user's browser and not the web application itself. In some situations, this might lead to unintentional exploitation by benign vectors [44] or prevent malicious payloads from executing, e.g. due to client-side XSS protection.

To address this problem, only intentionally malicious payloads are created, effectively eliminating true negatives. Every request that is able to bypass the WAF is considered a successful attack, while a blocked request is treated as a failed attack. We selected this approach because a browser oracle – where the payload actually has to be executed in the browser – depends on browser-specific behavior and might thus lead to incorrect results. These behaviors are relevant in a real-world scenario when trying to exploit a vulnerability, but negligible for our evaluation of XSS detection capabilities of WAFs. Ideally, the WAF blocks every payload that could lead to an exploit in any browser and in any context.

Treating the WAF's decision as an absolute also has drawbacks. Since we only create intentionally malicious payloads, we know that when the oracle does not detect an attack, it must be a false negative. However, false positives become a bigger challenge. When configuring a WAF, it is desirable to minimize false positives, which might lead to legitimate requests being blocked. Given the decision mode of our oracle (which treats every blocked request as a failed attack), simply blocking every request would give a perfect score in our evaluation. We therefore have to keep in mind that our evaluation method favors restrictive WAFs. In practice, this should not be a huge concern, because false positives are a common challenge for WAFs and default rules tend to be rather lenient as a result. In Section IV-A, we explain how different WAFs decide what constitutes an attack.

IV. Case Study

In this section, we provide the remaining details of our case study, including the tested SUTs, selected static attack lists (extracted from vulnerability scanners) for comparison and computing infrastructure that was used to execute the experiments. In Section IV-A, we describe the tested WAFs and briefly comment on differences between them. Section IV-B contains static lists of attack vectors used for comparison in this work. The computing infrastructure environment used to execute experiments is described in Section IV-C.

A. Web Application Firewalls

Three open source WAFs (ModSecurity [46], NAXSI [28] and lua-rest-waf [31]) were selected as targets, some of them in multiple configurations, resulting in a total of 7 SUTs in this case study. These applications were selected due to the availability of their source code and because they implement different request classification approaches, thus ensuring diversity.

We did not consider any commercial tools, WAFs that work exclusively based on whitelisting, or candidates that employ learning-based approaches. Deploying and running tests against WAFs based on whitelisting (which block all requests by default and only allow those that have been explicitly permitted) would have required the creation of application-specific whitelists. Similarly, learning-based WAFs would have required training before performing the evaluation, a process that was considered out of scope for this work.

WAF	Web server	Rule set	method	Parameter
ModSecurity 2	Apache 2.4	CRS 3.0 XSS	GET	q
ModSecurity 2	Apache 2.4	CRS 3.1 XSS	GET	q
ModSecurity 3	Apache 2.4	CRS 3.2 XSS	GET	q
lua-resty-waf	nginx 1.17	default XSS	GET	q
NAXSI	nginx 1.17	default	GET	q
NAXSI	nginx 1.17	wordpress whitelist	POST	comment
NAXSI	nginx 1.17	drupal whitelist	POST	user_mail

TABLE II: Target SUTs for evaluation.

Table II lists the selected WAFs, together with the set of rules, request method and payload parameter used for evaluation purposes.

The mode of operation is different for each of the WAFs listed in Table II. For this reason, each WAF and its respective configuration(s) are explained below.

a) ModSecurity: ModSecurity [46] is a WAF that can be enabled as a module in Apache [43] or nginx [14] and ships with a core rule set (CRS) [30]. The rules contained therein encode whether a request is classified as malicious or not (see Section III-C). The CRS is grouped into different attack categories (XSS, SQL injection, etc.). Only XSS rules were enabled in this case study. Each rule consists of a complex regular expression to detect malicious payloads in GET/POST HTTP parameters, cookies or other headers.

The CRS has a parameter called `paranoia level` (`PL`) that assumes integer values ranging from 1 to 4, which allows the administrator to select how strict the evaluation performed by ModSecurity should be. The default level for `PL` is 1, while a level of 2, according to [29],

> [...] is advised for moderate to experienced users who desire more complete coverage, and for all installations with elevated security requirements. PL2 may cause some FPs [False Positives] which you need to handle.

Levels 3 and 4 similarly result in an increasing number of false positives. In this work, only `PL 1` was considered to avoid a high rate of false positives. This is in line with the expected deployment options in practice, as additional adjustments to rules may be necessary when using higher levels [42].

b) NAXSI: NAXSI is a WAF that works as a module on nginx webservers [28]. In contrast to ModSecurity, NAXSI does not have complex patterns to match malicious requests, instead opting to blacklist single characters, e.g., "<". While this results in easily comprehensible rules, it may also lead to more false positives.

c) lua-resty-waf: This WAFs [31] works as a reverse proxy on top of the OpenResty stack [51], which is an application platform built on top of nginx. Similar to ModSecurity, rules consist of regular expressions.

Strength	# test cases	Reduction (in %)
2	99	99.97
3	794	99.76
4	4,766	98.60
5	19,311	94.35
6	58,251	82.97

TABLE III: Number of test cases in generated t-way test sets and reduction vs. exhaustive.

B. Static attack lists

In this case study, we wish to compare the attack vectors generated with CST against those used by state-of-the-art vulnerability scanners. Unfortunately, the tight coupling between test set generation and execution makes web vulnerability scanners such as Burp Suite [32] and OWASP ZAP [52] incompatible with our oracle and prototype tool. We therefore utilize three publicly available static lists of XSS attack vectors instead, which are of comparable quality to the vectors used by the aforementioned products.

a) rsnake: Robert Hansen, a widely known security researcher who also co-authored on a book on XSS exploits [16], created a list of XSS attack vectors [36] containing 73 handcrafted items, including a wide array of evasion techniques.

b) html5sec: Html5sec[3] is a cheat sheet that points out XSS issues in connection with attributes introduced in HTML5. We utilized a list of 136 attack vectors that try to exploit every issue mentioned therein [23].

c) portswigger: The portswigger list of attack vectors [37] was created by the cyber security company of the same name, commonly known as the developer of Burp Suite. In contrast to the other two lists, the 6047 entries in this list are not manually crafted, but instead generated by combining a set of HTML tags, attributes and payloads.

C. Experimental setup

All experiments were performed on a PC system running Arch Linux with an AMD Ryzen CPU and 16 GB memory. The total test generation time for all $t \in \{2, 3, 4, 5, 6\}$ using *CAgen* [49] was only 30 seconds. Table III lists the sizes of the generated combinatorial test sets as well as their reduction compared to an exhaustive test set (which would require $342,144$ test cases). Considering that we achieved approximately 120 requests/second during our evaluation, executing our combinatorial test sets for all strengths resulted in a cumulative elapsed time of about 12 minutes for each SUT.

V. Evaluation

This section presents the results of the evaluation of our case study. We introduce the evaluation metrics used to interpret and analyze the results in Section V-A. In Section V-B, we analyze the results achieved by the combinatorially generated XSS attack vectors. Finally, Section V-C contains a performance comparison between these vectors and their counterparts extracted from static lists.

[3] https://www.html5sec.org

A. Evaluation metrics

We use a slightly modified definition of EXPLOITATION RATE (ER) to quantify the performance of XSS attack vectors, a metric that is well-established in the literature [8], [41]. Specifically, for a given SUT and test set $\mathcal{T}$, the ER equals the ratio of all test cases (i.e. XSS attack vectors from the test set) that have received a test oracle verdict of *true* (i.e. successfully bypassing the WAF undetected). In other words, we obtain[4]:

$$\text{ER} = \frac{\#\text{ Attack vectors in } \mathcal{T} \text{ that bypass the WAF}}{\#\text{ Attack vectors in test set } \mathcal{T}}$$

When comparing the ER of two different test sets against the same SUT, a *higher* value is better.

B. Evaluation of CST results

We are interested in evaluating the achieved ERs for the generated combinatorial test sets with varying strengths from 2 to 6 against the different SUTs in the case study. The complete evaluation results for all targets and interaction strengths are given in Table V.

An important aspect is the influence of the interaction strength on the ER. The graphs in Figure 5 show that for each SUT, the exploitation rate is nearly constant for increasing strength. Although this may seem unimpressive at first, a constant ER for test sets with an increasing number of individual XSS attack vectors (i.e., test cases) also means more successful bypasses in absolute numbers for higher-strength test sets.

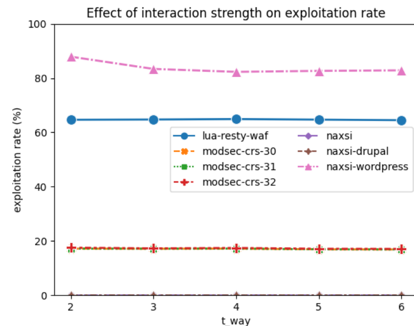

Fig. 5: Comparison of exploitation rates of combinatorial test sets for different strengths.

Two SUTs, "NAXSI" and "NAXSI-drupal" (NAXSI with Drupal whitelist), were able to block all attack vectors. This phenomenon arises from the unusual form of rules used by these WAFs. As explained in Section IV-A, NAXSI blacklists single characters, while all the other WAFs employ complex patterns in their rules. This leads to a high level of protection, as some characters that are common to many attack vectors (such as parenthesis, quotes or characters that start an escape sequence) are blocked. In a real-world scenario, decisions based on a single character are hardly enough to successfully detect a malicious request and will therefore yield

[4] They symbol # denotes the cardinality of a set.

a high number of false positives. This issue becomes very visible with the target "NAXSI-wordpress". The Wordpress whitelist allows several more characters in order to make sure benign requests are not blocked. However, this difference alone suffices to make this the SUT with the highest ER. In a real-world scenario, a WAF operating only on single character rules needs to work very closely with the underlying application to find a good balance between a high rate of true positives while still maintaining a low rate of false positives.

ModSecurity was the SUT with the second lowest ER for the pairwise test set of XSS attack vectors, but did not reject $\approx 17\%$ of the generated attack vectors. There was no difference in the value of the achieved ER depending on which version of the CRS had been used. Note that Table V reports a slightly lower number of total requests per combinatorial test set than what would be expected according to Table III for the SUT modsec-crs-32 (ModSecurity with CRS 3.2). This is because certain generated XSS attack vectors caused the web server to abruptly terminate the connection, leaving us unable to evaluate their effects. Based on an investigation of this issue, we suspect that this is a software bug that arises from the combination between these specific versions of the Apache web server and ModSecurity or an issue in building the relevant Docker container. However, this issue affects at most $\approx 2\%$ of test cases in each combinatorial test set. The observed behavior for this configuration of ModSecurity is otherwise similar to the results for other configurations.

C. Evaluation of static lists

Table VI shows the results for the static lists of attack vectors. Similar to our combinatorial approach, none of these vectors were able to bypass NAXSI or NAXSI-drupal, as all of them contained characters explicitly blocked by these two SUTs. As was the case for the combinatorial test sets, the number of successful injections is constant across all three ModSecurity SUTs.

Note that ERs of the portswigger list are either zero (in the case of lua-resty-waf, all three ModSecurity SUTs as well as NAXSI and NAXSI-drupal) or one (NAXSI-wordpress) in all cases. The reason for this rather unusual result is the minuscule amount of variation in this list of attack vectors.

D. Comparison between combinatorial and static test sets

Table IV lists the best ER achieved by a combinatorial test set in column `Best CT` and as well as the results achieved by the static lists. Note that the SUT modsec-crc-32 was excluded due to the observed irregularities explained above. As the results for all ModSecurity configurations are identical, they are listed as a single item.

In terms of ER, the combinatorial test sets outperform the static lists for the three SUTs lua-resty-waf, modsecurity-crc-30 and modsecurity-crc-31. For NAXSI-wordpress, the best CST result is weaker than static lists, but nevertheless still achieves a high ER of 87.9%.

Target	Best CT	rsnake	html5sec	portswigger
lua-resty-waf	64.9	6.8	7.4	0.0
ModSecurity	17.2	6.8	2.2	0.0
NAXSI-wordpress	87.9	97.3	95.6	100

TABLE IV: Exploitation rates in percent for the most important targets comparing the best combinatorial result with static lists.

target	strength	success	fail	total	ER (in %)
lua-resty-waf	2	64	35	99	64.6
modsec-crs-30	2	17	82	99	17.2
modsec-crs-31	2	17	82	99	17.2
modsec-crs-32	2	17	80	97	17.5
NAXSI	2	0	99	99	0.0
NAXSI-drupal	2	0	99	99	0.0
NAXSI-wordpress	2	87	12	99	87.9
lua-resty-waf	3	514	280	794	64.7
modsec-crs-30	3	136	658	794	17.1
modsec-crs-31	3	136	658	794	17.1
modsec-crs-32	3	136	652	788	17.3
NAXSI	3	0	794	794	0.0
NAXSI-drupal	3	0	794	794	0.0
NAXSI-wordpress	3	662	132	794	83.4
lua-resty-waf	4	3,093	1,673	4,766	64.9
modsec-crs-30	4	819	3,947	4,766	17.2
modsec-crs-31	4	819	3,947	4,766	17.2
modsec-crs-32	4	819	3,882	4,701	17.4
NAXSI	4	0	4,766	4,766	0.0
NAXSI-drupal	4	0	4,766	4,766	0.0
NAXSI-wordpress	4	3,922	844	4,766	82.3
lua-resty-waf	5	12,493	6,818	19,311	64.7
modsec-crs-30	5	3,265	16,046	19,311	16.9
modsec-crs-31	5	3,265	16,046	19,311	16.9
modsec-crs-32	5	3,265	15,800	19,065	17.1
NAXSI	5	0	19,311	19,311	0.0
NAXSI-drupal	5	0	19,311	19,311	0.0
NAXSI-wordpress	5	15,961	3,350	19,311	82.7
lua-resty-waf	6	37,565	20,686	58,251	64.5
modsec-crs-30	6	9,819	48,432	58,251	16.9
modsec-crs-31	6	9,819	48,432	58,251	16.9
modsec-crs-32	6	9,819	47,705	57,524	17.1
NAXSI	6	0	58,251	58,251	0.0
NAXSI-drupal	6	0	58,251	58,251	0.0
NAXSI-wordpress	6	48,259	9,992	58,251	82.8

TABLE V: Full evaluation results for CT suites.

VI. Related Work

Research has shown that applying XSS sanitization to input data is difficult and prone to errors [6], [38]. In [50], the authors provide an an overview of challenges to input sanitization and explore methods to this end available in web application frameworks. Even sophisticated dynamic approaches such as the method presented in [7] do not provide 100% security against such attacks. Several testing methods have been developed with the goal of bypassing such filtering mechanisms in order to improve XSS detection, including search-based testing [5], unit testing [26] and mutation-based testing [24], [39], as well as evolutionary approaches [12], [13], [45].

There has been research on SQL injection attacks with WAFs in place [1]–[3]. Estimates on the effectiveness of WAFs have been surveyed in [22].

There are several works in the literature relying on a CST approach for generating XSS attack vectors [18], establishing the applicability of CT to web application security testing. The

target	suite	success	fail	total	ER (in %)
lua-resty-waf	html5sec	10	126	136	7.4
modsec-crs-30	html5sec	3	133	136	2.2
modsec-crs-31	html5sec	3	133	136	2.2
modsec-crs-32	html5sec	3	132	135	2.2
NAXSI	html5sec	0	136	136	0.0
NAXSI-drupal	html5sec	0	272	272	0.0
NAXSI-wordpress	html5sec	130	6	136	95.6
lua-resty-waf	portswigger	0	6,047	6,047	0.0
modsec-crs-30	portswigger	0	6,047	6,047	0.0
modsec-crs-31	portswigger	0	6,047	6,047	0.0
modsec-crs-32	portswigger	0	6,047	6,047	0.0
NAXSI	portswigger	0	6,047	6,047	0.0
NAXSI-drupal	portswigger	0	12,094	12,094	0.0
NAXSI-wordpress	portswigger	6,047	0	6,047	100.0
lua-resty-waf	rsnake	5	68	73	6.8
modsec-crs-30	rsnake	5	68	73	6.8
modsec-crs-31	rsnake	5	68	73	6.8
modsec-crs-32	rsnake	5	68	73	6.8
NAXSI	rsnake	0	73	73	0.0
NAXSI-drupal	rsnake	0	146	146	0.0
NAXSI-wordpress	rsnake	71	2	73	97.3

TABLE VI: Full evaluation results for static attack vector lists.

techniques used to this end include the modelling of attack patterns found in XSS vulnerabilities [9] and the integration of constraints on the devised IPMs for a more strict generation of attack vectors [8]. Finally, an approach using locally optimized attack models is presented in [41], where the execution context was taken into account and locally optimized vectors were designed to specifically exploit certain structural properties occurring in an HTML page.

VII. Threats to Validity

In this section, we comment on possible threats to validity of this work.

First, with regards to internal validity, the primary observation is that we used a custom virtualized environment to execute our case study. This simulated network logically uses the same protocols that are also employed in an internet environment. However, we note that we have experienced unexpected behavior of one SUT (modsec-crs-32) during the execution of combinatorial test sets. Our chosen oracle does not allow to distinguish between false and true positives/negatives, and our developed grammar had to be quite conservative in order to not generate any true negatives, i.e. payloads that cannot lead to an exploit and are correctly identified as such. Since we considered any vector able to bypass a WAF as successful, this could lead to payloads skewing the results toward an overly high ER.

With regards to external validity, it is clear that the conducted case study is limited. However, we considered a diverse range of WAFs to make the resulting benchmark comparisons realistic.

VIII. Conclusion

The evaluation of the case study performed in this work shows that the exploitation rate of combinatorial test sets based on a dedicated attack grammar compares favorably to existing state-of-the-art attack sets. A practically constant ER for increasing strength of the combinatorial test sets implies an increase in the absolute number of successful attacks for increasing strength and that mere pairwise (i.e. strength $t = 2$) test sets lead to the successful bypass of some SUTs . In some cases, the combinatorial test tests outperformed static lists of XSS attack vectors considerably.

The evaluation further shows that while WAFs offer some additional protection against XSS injections to web applications, they are not infallible and may still leave web applications vulnerable to attacks. WAFs do offer some protection, even without much configuration, but they cannot substitute correct handling of input by the applications themselves.

The obtained results of the CST-approach for generating XSS attack vectors for bypassing WAFs in case study consisting of a diverse set of open source WAFs as SUTs lead to the conclusion that the CST approach has to be considered as delivering state-of-the-art performance when testing for vulnerabilities in WAFs in addition to its previous success reported in the literature in (direct) XSS attacks on web applications.

Although our results show that the CST approach proposed in this work compares favorably to traditional approaches for bypassing WAFs, there are avenues for future research. It is clear that the integration of a more sophisticated oracle would enhance the overall approach. Moreover, in future research, the CST approach proposed in this work could be refined and used to develop exploits specifically targeting a particular WAF and a single web application.

Disclaimer: Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.

References

[1] Dennis Appelt, Cu D. Nguyen, and Lionel Briand. Behind an Application Firewall, Are We Safe from SQL Injection Attacks? In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, Graz, Austria, April 2015. IEEE. http://ieeexplore.ieee.org/document/7102581/.

[2] Dennis Appelt, Cu D. Nguyen, Annibale Panichella, and Lionel C. Briand. A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls. *IEEE Transactions on Reliability*, 67(3):733–757, September 2018. https://ieeexplore.ieee.org/document/8395015/.

[3] Dennis Appelt, Annibale Panichella, and Lionel Briand. Automatically Repairing Web Application Firewalls Based on Successful SQL Injection Attacks. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 339–350, Toulouse, October 2017. IEEE. http://ieeexplore.ieee.org/document/8109099/.

[4] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security Privacy*, 3(1):84–87, 2005.

[5] Andrea Avancini and Mariano Ceccato. Security Testing of Web Applications: A Search-Based Approach for Cross-Site Scripting Vulnerabilities. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 85–94, Williamsburg, VA, USA, September 2011. IEEE. http://ieeexplore.ieee.org/document/6065200/.

[6] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (Sp 2008)*, pages 387–401, Oakland, CA, USA, May 2008. IEEE. http://ieeexplore.ieee.org/document/4531166/.

[7] Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137, pages 23–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. http://link.springer.com/10.1007/978-3-540-70542-0_2.

[8] Josip Bozic, Bernhard Garn, Ioannis Kapsalis, Dimitris Simos, Severin Winkler, and Franz Wotawa. Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 207–212, Vancouver, BC, Canada, August 2015. IEEE. http://ieeexplore.ieee.org/document/7272934/.

[9] Josip Bozic, Dimitris E. Simos, and Franz Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test - AST 2014*, pages 1–7, Hyderabad, India, 2014. ACM Press. http://dl.acm.org/citation.cfm?doid=2593501.2593502.

[10] B. Chess and B. Arkin. Software security in practice. *IEEE Security Privacy*, 9(2):89–92, 2011.

[11] John D Day and Hubert Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[12] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean-Luc Richier. XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 815–817, Montreal, QC, Canada, April 2012. IEEE. http://ieeexplore.ieee.org/document/6200193/.

[13] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. KameleonFuzz: Evolutionary fuzzing for black-box XSS detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy - CODASPY '14*, pages 37–48, San Antonio, Texas, USA, 2014. ACM Press. http://dl.acm.org/citation.cfm?doid=2557547.2557550.

[14] F5, Inc. NGINX. https://www.nginx.com/, January 2021. Accessed: 2021-01-13.

[15] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. volume 101 of *Advances in Computers*, pages 1 – 51. Elsevier, 2016.

[16] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, Petko Petkov, and an O'Reilly Media Company. Safari. XSS Attacks. https://go.oreilly.com/queensland-university-of-technology/library/view/-/9780080553405/?ar, 2011.

[17] Jose Fonseca, Marco Vieira, and Henrique Madeira. Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 365–372, Melbourne, Australia, December 2007. IEEE. http://ieeexplore.ieee.org/document/4459684/.

[18] Bernhard Garn, Ioannis Kapsalis, Dimitris E. Simos, and Severin Winkler. On the applicability of combinatorial testing to web application security testing: A case study. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing - JAMAICA 2014*, pages 16–21, San Jose, CA, USA, 2014. ACM Press. http://dl.acm.org/citation.cfm?doid=2631890.2631894.

[19] Bernhard Garn, Marco Radavelli, Angelo Gargantini, Manuel Leithner, and Dimitris E. Simos. A Fault-Driven Combinatorial Process for Model Evolution in XSS Vulnerability Detection. In Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali, editors, *Advances and Trends in Artificial Intelligence. From Theory to Practice*, volume 11606, pages 207–215. Springer International Publishing, Cham, 2019. http://link.springer.com/10.1007/978-3-030-22999-3_19.

[20] Shashank Gupta and B. B. Gupta. Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8(S1):512–530, January 2017. http://link.springer.com/10.1007/s13198-015-0376-0.

[21] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 281–296, Baltimore, MD, August 2018. USENIX Association.

[22] Hannes Holm and Mathias Ekstedt. Estimates on the effectiveness of web application firewalls against targeted attacks. *Information Management & Computer Security*, 21(4):250–265, January 2013. https://doi.org/10.1108/IMCS-11-2012-0064.

[23] Paweł Krawczyk and Heiderich Mario. HTML5sec-Injections-Jhaddix.txt. https://github.com/danielmiessler/SecLists/blob/bb915befb208fd900592bb5a25d0c5e4f869f8ea/Fuzzing/HTML5sec-Injections-Jhaddix.txt, 2019. Accessed: 2020-10-31.

[24] Qing Li, Jinfu Chen, Yongzhao Zhan, Chengying Mao, and Huanhuan Wang. Combinatorial Mutation Approach to Web Service Vulnerability Testing Based on SOAP Message Mutations. In *2012 IEEE Ninth International Conference on E-Business Engineering*, pages 156–162, Hangzhou, China, September 2012. IEEE. http://ieeexplore.ieee.org/document/6468233/.

[25] Mirakhorli, Mehdi and Galster, Matthias and Williams, Laurie. Understanding software security from design to deployment. *SIGSOFT Softw. Eng. Notes*, 45(2):25–26, April 2020.

[26] Mahmoud Mohammadi, Bill Chu, Heather Richter Lipford, and Emerson Murphy-Hill. Automatic web security unit testing: XSS vulnerability detection. In *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*, pages 78–84, Austin, Texas, 2016. ACM Press. http://dl.acm.org/citation.cfm?doid=2896921.2896929.

[27] Nabil M. Mohammed, Mahmood Niazi, Mohammad Alshayeb, and Sajjad Mahmood. Exploring software security approaches in software development lifecycle: A systematic mapping study. *Computer Standards & Interfaces*, 50:107 – 115, 2017.

[28] NBS System. Naxsi. https://github.com/nbs-system/naxsi, October 2020. Accessed: 2020-10-31.

[29] OWASP. FAQ – OWASP ModSecurity Core Rule Set. https://coreruleset.org/faq/, 2020. Accessed: 2020-11-08.

[30] OWASP. ModSecurity Core Rule Set. https://coreruleset.org/, 2020. Accessed: 2020-10-31.

[31] p0pr0ck5. P0pr0ck5/lua-resty-waf. https://github.com/p0pr0ck5/lua-resty-waf, October 2020. Accessed: 2020-10-31.

[32] PortSwigger Ltd. Burp Suite. https://portswigger.net/burp, January 2021. Accessed: 2021-01-13.

[33] B. Potter and G. McGraw. Software security testing. *IEEE Security Privacy*, 2(5):81–85, 2004.

[34] Rahman, Akond and Williams, Laurie. A bird's eye view of knowledge needs related to penetration testing. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, HotSoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Kenneth Reitz. Requests: HTTP for Humans. https://requests.readthedocs.io/en/master/, 2020. Accessed: 2020-10-31.

[36] rsnake. XSS-RSNAKE.txt. https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/XSS/XSS-RSNAKE.txt, 2019. Accessed: 2020-10-31.

[37] s7x. XSS-Cheat-Sheet-PortSwigger.txt. https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/XSS/XSS-RSNAKE.txt, 2019. Accessed: 2020-10-31.

[38] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*, page 587, Chicago, Illinois, USA, 2011. ACM Press. http://dl.acm.org/citation.cfm?doid=2046707.2046775.

[39] Hossain Shahriar and Mohammad Zulkernine. MUTEC: Mutation-based testing of Cross Site Scripting. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 47–53, Vancouver, BC, Canada, May 2009. IEEE. http://ieeexplore.ieee.org/document/5068458/.

[40] D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker. Combinatorial Methods in Security Testing. *Computer*, 49(10):80–83, 2016.

[41] Dimitris E. Simos, Bernhard Garn, Jovan Zivanovic, and Manuel Leithner. Practical Combinatorial Testing for XSS Detection using Locally Optimized Attack Models. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 122–130, Xi'an, China, April 2019. IEEE. https://ieeexplore.ieee.org/document/8728914/.

[42] Jatesh Jagraj Singh, Hamman Samuel, and Pavol Zavarsky. Impact of Paranoia Levels on the Effectiveness of the ModSecurity Web Application Firewall. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 141–144, South Padre Island, TX, April 2018. IEEE. https://ieeexplore.ieee.org/document/8367754/.

[43] The Apache Software Foundation. Apache HTTP Server Project. https://httpd.apache.org/, January 2021. Accessed: 2021-01-13.

[44] Michael Thelin. Web Security: Cross-site scripting attacks using UTF-7. http://michaelthelin.se/security/2014/06/08/

web-security-cross-site-scripting-attacks-using-utf-7.html, 2014. Accessed: 2020-11-22.

[45] Omer Tripp, Omri Weisman, and Lotem Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, page 347, Lugano, Switzerland, 2013. ACM Press. http://dl.acm.org/citation.cfm?doid=2483760.2483776.

[46] Trustwave Spiderlabs. ModSecurity: Open Source Web Application Firewall. https://modsecurity.org/, 2020. Accessed: 2020-10-31.

[47] I. A. Tøndel, M. G. Jaatun, and J. Jensen. Learning from software security testing. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 286–294, 2008.

[48] Andrew van der Stock, Brian Glas, Neil Smithline, and Torsten Gigler. OWASP Top 10 2017. https://owasp.org/www-project-top-ten/2017/. Accessed: 2020-10-25.

[49] Michael Wagner, Kristoffer Kleine, Dimitris E. Simos, Rick Kuhn, and Raghu Kacker. CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 191–200, Porto, Portugal, October 2020. IEEE. https://ieeexplore.ieee.org/document/9155722/.

[50] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, volume 6879, pages 150–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. http://link.springer.com/10.1007/978-3-642-23822-2_9.

[51] Yichun Zhang. OpenResty. https://openresty.org/en/, January 2021. Accessed: 2021-01-13.

[52] ZAP Dev Team. OWASP Zed Attack Proxy (ZAP). https://www.zaproxy.org/, January 2021. Accessed: 2021-01-13.