A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS) Vulnerabilities in Web Application Security Testing

Dimitris E. Simos[1](✉), Kristoffer Kleine[1],
Laleh Shikh Gholamhossein Ghandehari[2], Bernhard Garn[1], and Yu Lei[2]

[1] SBA Research, 1040 Vienna, Austria
{dsimos,kkleine,bgarn}@sba-research.org
[2] Department of Computer Science and Engineering,
University of Texas at Arlington, Arlington, TX 76019, USA
laleh.shikhgholamhosseing@mavs.uta.edu, ylei@cse.uta.edu

Abstract. Web applications typically employ sanitization functions to sanitize user inputs, independently whether this input is assumed to be legitimate, invalid or malicious. When such functions do not work correctly, a web application immediately becomes vulnerable to security attacks such as XSS. In this paper, we report a combinatorial approach to analyze XSS vulnerabilities in web applications. Our approach first performs combinatorial testing where a set of test vectors is executed against a subject application. If one or more XSS vulnerabilities are triggered during testing, we analyze the structure of each test vector to identify XSS-inducing combinations of its parameter model. If an attack vector contains an XSS-inducing combination, then the execution of this vector will successfully exploit an XSS vulnerability. Identification of XSS-inducing combinations provides insights about which kinds of user input might still be leverageable for XSS attacks and how to correct the function to provide better security guarantees. We conducted an experiment in which our approach was applied to four sanitization functions from the Web Application Vulnerability Scanner Evaluation Project (WAVSEP). The experimental results show that our approach can effectively identify XSS-inducing combinations for these sanitization functions.

Keywords: Combinatorial testing · XSS · Fault localization · Security testing

1 Introduction

Web application security is as important as ever but pervasive ubiquitous computing, bundled with 24/7 network access, makes any connected web application especially susceptible to attacks. Naturally, injection attacks are remote exploits

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing AG 2016. All Rights Reserved
F. Wotawa et al. (Eds.): ICTSS 2016, LNCS 9976, pp. 70–85, 2016.
DOI: 10.1007/978-3-319-47443-4_5

which can cause security breaches. Cross-site scripting (XSS) falls into this category and constitutes the third serious vulnerability according to the Open Web Application Security Project (OWASP) [22]. We focus on analyzing XSS vulnerabilities where we distinguish between two different types of XSS, namely reflected XSS and stored XSS. In the former case the web server response contains some data from the corresponding request, while the latter case includes data stored permanently on the server (e.g., in a database). In line of this work we are concerned only for reflected XSS vulnerabilities.

In this paper, we apply for the *first time* a fault-localization technique based on combinatorial methods to identify one or more combinations of input parameter values that would definitely trigger an XSS vulnerability for a given system under test (SUT). We refer to these combinations as XSS-inducing combinations or simply inducing combinations. If an XSS attack vector (test vector) contains an inducing combination, then the execution of this test vector against the SUT will successfully exploit an XSS vulnerability. The identification of inducing combinations provides important information about why an input filter fails to sanitize a malicious vector, which in turns helps to make necessary corrections.

Note that this is different from traditional fault localization, which is aimed at identifying the location of a fault in the source code. Sanitization functions are typically employed in web applications to sanitize invalid or malicious user inputs. XSS vulnerabilities, if they exist, are in most cases contained in these sanitization functions, which are mostly simply referred to as *filters*. Thus, the location of an XSS vulnerability in the source code is typically considered known or not difficult to be identified. However, designing and implementing rigorous and secure input filters is a very complicated and challenging task [1]. In particular, when an input filter does not work as expected, it could be difficult for one to understand why it does not work and how to correct a vulnerable filter. The results of this paper enhance the capabilities of security testers to design better attack models for web applications but at the same time guide the developers on how to improve the filtering mechanisms met in such applications.

In Sect. 2 we describe related work for web application security testing and fault localization techniques. Sections 3 and 5 reviews past achievements on combinatorial testing for web security testing and fault localization methods, respectively, that relate to this work. Section 4 discusses the test execution method used in this work. In Sect. 6 we present our methodology for analyzing XSS vulnerabilities using combinatorial based fault localization methods. An experimental evaluation that validates our approach is given in Sect. 7. Finally, Sect. 8 concludes the work and discusses directions for future work.

2 Related Work

In this section, we describe related works with respect to fault localization approaches for combinatorial testing and security testing frameworks devoted to XSS detection. For a systematic literature review on research devoted to XSS we refer to [12] while for important contributions in combinatorial testing and fault localization that relate to the work presented in this paper we refer to Sects. 3 and 5, respectively, and cited references there in.

Web Application Security Testing Frameworks. Security testing is meant to support vulnerability detection, and for this task several approaches and tools have been developed in the past. In the following, we depict the most important of them. A comparison of several penetration testing tools is given in [7,15]. The authors of these works compare commercial as well as open source penetration testing tools by testing several web applications. Security testing tools incorporating fuzzing techniques have been presented in [5,6,20]. The authors of the last two works apply evolutionary approaches and learning in order to detect potential vulnerabilities. Even though these works add towards test automation, complete automation of the security testing process remains a very active challenge. Recent works on XSS vulnerability detection include unit testing methods that can detect XSS vulnerabilities which cannot be found by static analysis tools [16] and attack patterns for black-box security testing of web applications [19].

It is evident that even though a lot of works have been devoted to XSS vulnerability detection very few of them focus on analyzing these vulnerabilities and even fewer correlate malicious vectors with sanitizing functions.

Fault Localization Techniques Based on Combinatorial Methods. Combinatorial testing has been shown to be a very effective testing strategy [13]. A t-way combinatorial test set is designed to detect failures that are triggered by combinations involving no more than t parameters. After a failure is detected, the next task is to identify the fault that causes the failure. The problem of fault localization can be divided into two sub-problems: (1) Identifying failure-inducing combinations. A combination is failure-inducing or simply inducing if its existence in a test causes the test to fail. (2) Identifying actual faults in the source code. A fault is a code defect that can be an incorrect, extra, or missing statement. As explained in Sect. 1, we are mainly interested in identifying XSS-inducing combinations. Thus, in the following, we will focus on existing approaches to identify failure inducing combinations.

Two techniques, called `FIC` and `FIC_BS` [24], take as input a single failed test from a combinatorial test set, and identify as output a minimal inducing combination that causes the test to fail. The main idea of the two techniques consists of changing, in a systematic manner, the parameter values in the failed test. A parameter value is considered to be involved in an inducing combination if changing it to a different value causes the failed test to pass. It is assumed that changing a parameter value does not introduce any new inducing combination.

The `AIFL` technique in [18] first identifies a set of suspicious combinations as candidates for being inducing. Second, it generates a group of tests for each failed test. After executing the newly generated tests, combinations which appeared in the passed tests are removed from the suspicious set. The `IterAIFL` technique is an iterative approach proposed by Wang et al. in [21]. It iteratively generates and refines suspicious set until it becomes stable.

In our earlier work, we developed a approach called BEN that identifies suspicious combinations in the same way as `AIFL` and `IterAIFL`. However, BEN produces a ranking of suspicious combinations and focuses on the most suspicious combinations. Moreover, BEN significantly differs from `AIFL` and `IterAIFL` in the way of generating new tests. A detailed description of BEN is given in Sect. 5.

Lastly, to the best of our knowledge this is the first work where combinatorial based fault localization techniques are applied to analyze security vulnerabilities.

3 Combinatorial Testing for Web Security Testing

Combinatorial testing has been successfully applied for testing (critical) software systems in large organizations [11]. It is an already proven method for black-box security testing of large-scale web software systems [2,3,7] where t-way testing was applied successfully to XSS detection. In this section, we review these key contributions in web security testing that are based on combinatorial methods and are used as a basis for analyzing XSS vulnerabilities via fault localization methods throughout this paper. For a general treatment of the field of combinatorial testing we refer the interest reader to the surveys of [4,17].

Throughout this paper, we are uniformly using a strength four ($t = 4$) test set against the SUTs for reasons explained later in this section. The underlying combinatorial model of XSS attack vectors is a refined and extended version from the works in [2,7] and is a form of input parameter model [10]. Its goal is to discretize the input space to parameters and discrete values so that these can be given to combinatorial testing tools.

The generated test vectors aim at producing valid JavaScript code when these are executed against SUTs. A description of parameters that appear in the input model has briefly been mentioned in [2,7], however we give an excerpt here, for the sake of completeness:

– The **JSO** (JavaScript Opening Tags) type represents tags that open a JavaScript code block.
– The **WS** (white space) type family represents white space characters.
– The **INT** (input termination) type represents values that terminate the original valid tags (HTML or others).
– The **EVH** (event handler) type contains values for JavaScript event handlers.
– The **PAY** (payload) type contains executable JavaScript.
– The **PAS** (payload suffix) type contains different values that should terminate the executable JavaScript payload (PAY parameter).
– The **JSE** (JavaScript end tag) type contains different forms of JavaScript end tags.

Moreover, this input model is optimized to fit to the employed test execution method (see Sect. 4). A suitable metric has been introduced in [2] to assess the quality of produced combinatorial test sets for XSS detection, called exploitation rate (ER), which measures the proportion of XSS attack vectors that were successful, e.g. the ones that exploit an XSS vulnerability, per given test set and SUT.

In particular, past work of ours has revealed that the usage of a 4-way test set (with constraints) yields satisfactory practical results for web application security testing and is justified as follows:

- In the majority of our past security testing experiments [2,3,7] we have witnessed that higher strength interaction testing yields better results w.r.t. exploitation rate. More specifically, we were able to report an increase in the exploitation rate when moving from 2-way to 3-way and 4-way testing. Also, in [2] we reported on cases where only a 4-way test set was able to successfully trigger XSS exploits for specific SUTs but none of the test sets with weaker t-way coverage properties could.
- In some of our past experiments [7], we have noticed performance issues when moving from pairwise-testing to higher interaction testing. Depending on the test execution method (see Sect. 4) (i.e. used penetration testing tool), the SUT (i.e. tested HTTP parameter of a web application) and the underlying operating system, we have seen execution times to vary greatly between repeated test runs. We further noticed SUTs to become unresponsive, as well as, increased memory usage. However, we were still able to exploit XSS vulnerabilities using a 4-way test set.
- An important finding in the post-processing of 2-way test sets used in [3] was that it revealed a surprising high percentage of 3-way and 4-way combinations covered in the successful XSS attack vectors (per test set and SUT).

These statements are in accordance with a relationship known as *interaction rule* in combinatorial testing which is based on empirical data and shows that most software faults are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six [14].

4 Penetration Testing Execution Methods

In this section, we provide details about the penetration testing execution method we have used in our experimental evaluation. We give a detailed description of its procedure, functionality and test oracle, applicable when testing for XSS vulnerabilities. The described method can be applied to security testing in general, but in this paper we focus explicitly on penetration testing, e.g. exploiting XSS vulnerabilities, where the main difference (to security testing) relies on the fact that we initiate the testing procedure once the web applications are installed in an operational environment. The main difference to conventional penetration testing is that we are not interested in pinpointing *where* a vulnerability is located in the source code, but rather to analyze a known vulnerable input field in a web application in order to get insights into its structure, i.e. the necessary degree of interaction to trigger the successful exploitation of an XSS vulnerability.

Test Execution. As test execution environment we used the Burp Suite[1] which is an integrated platform for performing security testing of web applications. It is widely used by security professionals since it allows to perform many penetration testing tasks.

In our case the Intruder module of BURP was used to execute our test vectors. Intruder offers automated customized attacks against web applications, to identify and exploit all kinds of security vulnerabilities including XSS attacks. In order to test an SUT we supplied its location (server, port and URL) to Intruder and also provided the position for the input parameter. Then, our test set consisting of XSS attack vectors was loaded and executed one by one. The response (HTML) of the SUT for each test vector was recorded and supplied to the test oracle in order to determine whether an XSS vulnerability was triggered.

Test Oracle. The usual penetration testing procedure is mostly concerned with finding which parts of a web application are potentially vulnerable to an XSS attack. Here, the tester submits a request with user-controlled string in a HTTP-parameter (e.g. the user enters a string <`script`> in a search function and submits the query) and then examines the HTML response page from the web application whether it contains any part of the submitted string. If there are no sanitization functions invoked on the input at all, then this input field is a very probable candidate for having an XSS vulnerability. It is a common practice in security testing to rely on string matching as the underlying test oracle which is commonly referred to as *reflection oracle*. This process is repeated with all HTTP parameters in a web application. However, the reflection oracle can not decide whether an identically reflected user input string would actually be executed by a web browser. Therefore, the reflection oracle decision is not indicative of the vector actually triggering an XSS vulnerability. Thus, in relation to the detection of true XSS an oracle relying on reflection alone is not infallible as it suffers both from false positives and false negatives. In order to determine if the XSS vulnerability was indeed triggered by a test vector – meaning that we have a true XSS – the response of the web application needs to be evaluated under real-world conditions.

This necessary task can be fulfilled by employing a new test oracle, henceforward called the *execution oracle*. As indicated by the name, this oracle operates similar to a web browser and evaluates/parses the page response from a web application. The generated test vectors must be designed in such a way that their behavior is detectable by the execution oracle. Additionally, in the presented form it must be ensured that this behavior is distinct from normal intended behavior by the SUT, so that we can deduce true XSS by page-parsing anomaly detection. We have used the XSS Validator extension of BURP to fill the role of the execution oracle. The inner workings of the Validator are described below in detail. We state an important fact: Under these conditions, every vector marked as triggering by the execution oracle is indeed a test vector which triggers a true XSS vulnerability exploitation and as such the execution oracle

[1] http://portswigger.net/burp/.

does not produce false positives. To illustrate this point, consider the test vector `onError=alert(1)` which is reflected inside the body tag of an HTML page. Under the assumption that the web application applies no filtering to the input then this vector will be reflected without changes in the page response and the reflection oracle will flag this vector. The execution oracle however, will not flag this vector because it does not exploit the vulnerability.

An instantiation of the execution oracle can be found in the XSS Validator[2] extension to BURP. This extension enhances the test execution capabilities of BURP by adding a detection mechanism of triggered XSS vulnerabilities.

The XSS Validator receives the response from the SUT (including the reflected test vector) and renders the HTML. During rendering, JavaScript contained in the website will be executed. When it is detected that JavaScript was executed which originated from a test vector then this test vector is flagged as having triggered the XSS vulnerability.

Since the Validator extension comes with its own set of test vectors and is targeted towards the detection of XSS vulnerabilities triggered by them we modified the code to use our own test vectors and adapted the detection code to recognize behavior triggered by them (see Sect. 6.2 for more details).

5 Fault Localization Based on Combinatorial Methods

BEN [8,9] adopts a spectrum-based fault localization technique and has been applied to a Siemens test set and two programs i.e., grep and gzip. It leverages the results of the combinatorial test set and generates the ranking of statements in terms of their likelihood of being faulty. BEN consists of two major phases: (1) In phase 1, BEN identifies a combination that is very likely to be a failure-inducing combination. (2) In phase 2, BEN takes the failure-inducing combination identified in phase 1 and then produces a ranking of statements in the source code by analyzing the spectra of the small group of tests.

In this work, we only applied the first phase of BEN because we are not interested on the ranking of statements in the source code since we are following a black-box security testing approach. Therefore, we focus solely on the first phase, identifying failure-inducing combinations. BEN takes the input parameter model and a t-way combinatorial test set with execution results as input, and adopts an iterative framework to identify inducing combinations of size t or larger. At each iteration, BEN analyzes a test set F, which initially is the t-way combinatorial test set taken as input. BEN first identifies a set of t-way suspicious combinations, π, then, ranks them based on their suspiciousness, i.e., likelihood to be inducing.

Next, a small set of new tests, F', is generated. If all tests in F' that contain a suspicious combination c are failing, then c is marked as an inducing combination, and the process stops. Otherwise, all tests in F', will be added to test set F, to refine the set of suspicious combinations and their ranking. BEN continues

[2] https://portswigger.net/bappstore.

the two steps, i.e., rank and test generation iteratively until a t-way suspicious combination is marked as an inducing combination or a stopping condition is satisfied [9]. In the latter case, no t-way inducing combination is identified, BEN increases the size of inducing combination, and tries to identify a $(t + 1)$-way inducing combination.

Rank generation and test generation are based on two notions, suspiciousness of a combination and suspiciousness of the environment of a combination. Informally, the environment of a combination consists of other parameter values that appear in the same test case. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. Moreover, new tests are generated for the most suspicious combinations. Let f be a new test generated for a suspicious combination c. Test f is generated such that it contains c and the suspiciousness of the environment for c is minimized. If f fails, it is more likely to be caused by c instead of other values in f.

This process is repeated until an inducing combination is found. Note that this process must terminate, as a failed test is by definition an inducing combination. Note that if there is a resource limitation, the user can stop the process. The top-ranked suspicious combination is reported as failure-inducing combination, in this case.

6 Methodology

In this section, we present our approach for analyzing XSS vulnerabilities using combinatorial based fault localization methods. Our methodology is comprised of two parts: First executing XSS attack vectors against SUTs and second identifying one or more combinations of input values that can trigger a successful XSS exploit. Our utter goal is to map the failure-inducing combinations found to XSS-inducing combinations. As explained in Sects. 4 and 5, respectively, we used the BURP suite for the first part and the BEN tool for the second. Further, we discuss below modifications of the BEN tool needed for XSS detection and also the necessity for a refinement of the attack model.

6.1 Modifications of BEN for XSS Detection

As explained in Sect. 5, BEN first looks for a t-way inducing combination, where t is the strength of the initial test set. Since all t-way combinations are covered by the t-way combinatorial test set, BEN guarantees to identify t-way inducing combination if such a combination exists. When there is no t-way inducing combination, BEN looks for $(t+1)$-way inducing combination that is covered by the t-way test set.

For our experiments, we modified BEN to take the size of inducing combination as well as the t-way combinatorial test set. The user can search for an inducing combination whose size is equal to, greater or less than the strength of combinatorial test set, t. When the size of inducing combination is equal to or less than t, BEN could identify inducing combination of a requested size, if there is any. When the size of inducing combination is greater than the strength of the combinatorial test set, BEN starts looking for an inducing combination with the requested size is covered by the test set. In this case, BEN does not search for t-way inducing combination, although it may exist.

In the test generation step, a set of new tests is generated for a user-specified number of top-ranked suspicious combinations. Note that the user could specify the number of top-ranked suspicious combinations and the number of tests generated for each top-ranked combination. The more tests generated, the more effort it takes to execute them, but the more confidence we have about the identified inducing combinations. Moreover, the bigger the top-ranked set, the more effort to generate and execute the new tests, but the faster an inducing combination may be identified. This is because if an inducing combination c is included in the top-ranked set, c is identified to be an inducing combination in the first iteration. Otherwise, it may take multiple iterations for c to move up into the top-ranked set.

For our experiments, we configure BEN to generate two tests for each of the five top ranked suspicious combinations at each iteration. So, at each iteration maximum 10 new tests will be added to the test set. Note that this is a practical decision made in consideration with resource constraints.

6.2 Model Refinement

We revised our combinatorial model of XSS attack vectors from [2,7] to fit to the new execution oracle based on the Validator extension of BURP and to limit the size of the generated 4-way test set (due to the performance issues mentioned in Sect. 3), resulting in a significantly more robust and sophisticated test framework able to cast a 100 % confidence decision on triggering test cases. To this end, we changed some parameter values and removed some others such that the resulting test vectors are in line with the implementation of hooks in the Validator so we can detect triggering test vectors. Most importantly, we chose two kinds of values for the payload parameter. One kind contains a call to the built-in JavaScript `alert` function while the other defines the `src` attribute pointing to some predefined and non-existing resource. Both types of these payloads trigger detectable behavior at runtime by PhantomJS. We have also verified this kind of JavaScript is not contained in our SUTs used in the experimental evaluation.

7 Experimental Evaluation

In this section, we conduct an experimental evaluation in order to validate our methodology for analyzing XSS vulnerabilities.

7.1 Design of the Experiment

The purpose of the experiment is to have a setup where we can evaluate our methodology for analyzing XSS vulnerabilities using combinatorial based fault localization techniques. To this end we choose 4 input fields as SUTs from WAVSEP, the Web Application Vulnerability Scanner Evaluation Project[3], version 1.2. WAVSEP is a web application specifically designed to allow testing for various kinds of XSS exploits, among other vulnerabilities. In contrast with training applications for web application security testing that have been thoroughly tested in the past [12], WAVSEP offers sophisticated filter mechanisms and the majority of its SUTs can be tested for XSS vulnerabilities. In the following, we give details about the chosen input fields.

In particular, we use four input HTTP parameters as SUTs out of the WAVSEP when testing for XSS vulnerabilities. Each SUT receives over HTTP one GET parameter which is reflected on the page in different contexts. Also, the input might optionally be filtered by a SUT specific sanitization function.

SUT ID	SUT name	Reflection site
1	Tag2HtmlPageScope	`<body>$input</body>`
2	Tag2TagStructure	`<input type="text" value="$input">`
3	Event2TagScope	`<img src="$input">`
4	Event2DoubleQuotePropertyScope	`<img src="$input">`

We give an description of these four SUTs, below:

SUT 1. This SUT just outputs the received parameter without modifications into the HTML body tag. Thus, possible exploits could just inject any HTML tag without having to worry about properly terminating a preceding tag in the page.

SUT 2. This SUT outputs the received parameter without modifications into the `value` attribute of an input tag.

SUT 3. This SUT outputs the received parameter into the `src` attribute of an image tag and filters angle brackets.

SUT 4. This SUT outputs the received parameter into the `src` attribute of an image tag and filters angle brackets and single quotes.

Test Vectors. We have employed the `ACTS` combinatorial test generation tool [23] for automated test generation of test vectors. The tool is developed jointly by the US National Institute of Standards and Technology and the University of Texas at Arlington and currently has more than 1400 individual and corporate users. In line of this work, we generated a 4-way test set consisting of 6891 test vectors.

Workflow. The test vectors described above were then all executed against all four SUTs and classified as either triggering an XSS vulnerability or not. Then,

[3] https://github.com/sectooladdict/wavsep.

BEN was run on the abstract test set together with the positions of vectors which did trigger a vulnerability (positive vectors) one time for each SUT. In the first round BEN searched for 4-way suspicious combinations and produced a set of recommended tests. These tests were then translated to concrete attack vectors and executed again. Depending on the result, BEN classified the underlying suspicious combinations either as inducing (in the case of all recommended tests succeeding) or not.

In the case that inducing combinations were found, we instructed BEN to look for lower strength faults to confirm if the fault was a true 4-way fault or an embedded lower strength fault. In the other case, when not all recommended tests succeeded, we instructed BEN to look for 5 or 6-way inducing combinations.

7.2 Results and Analysis

Here, we present our evaluation results grouped per analyzed SUT. In particular, we evaluate our findings w.r.t. underlying vulnerabilities and also correlate failure-inducing combinations with shortcomings in the filter mechanisms.

SUT 1. The initial test execution revealed 24 test vectors to trigger the XSS vulnerability. All ten recommended tests produced by BEN for 4-way suspicious combinations did not trigger the vulnerability. Therefore, we increased the strength and searched for 5 and 6-way suspicious combinations. As all eight recommended tests from 6-way suspicious combinations triggered the vulnerability we arrived at four inducing combinations of strength 6. In Table 1 we show the composition of these recommended tests and highlight in red the inducing combinations.

In the table the common structure of the triggering vectors is clearly visible as they all start with an opening `img` tag and contain a reference to the predefined resource. The other components of the inducing combination make sure that the vector does not contain any interfering characters to ensure that the vector will be parsed correctly when reflected in the page response.

Table 1. Recommended tests with embedded 6-way inducing combinations

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`\>`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`'\>`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	ϵ
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`<</script>`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`>`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`>>`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`</script>">`
`<img`	ϵ	␣	ϵ	ϵ	ϵ	`src="invalid"`	ϵ	ϵ	ϵ	`</script>`

Since this SUT applies no filter to the input parameter the final page response will include the following HTML body when the first recommended test in the table above is submitted:

```
<body><img src="invalid"\></body>
```

This will force the application to load the resource `invalid` and thus trigger the XSS vulnerability.

SUT 2. The initial test execution revealed 3 test vectors to trigger the XSS vulnerability. Four out of ten generated recommended tests derived from 4-way suspicious combinations triggered the vulnerability. Because of this we instructed BEN to search for 5-way inducing combinations. Since all four recommended tests for 5-way suspicious combinations triggered, we found two inducing combinations. The final recommended tests and inducing combinations are summarized in Table 2.

Table 2. Recommended tests with embedded 5-way inducing combinations

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
`"><script>`	␣	ϵ	␣	ϵ	␣	`alert(1)`	␣	ϵ	␣	`</script>">`
`"><script>`	␣	ϵ	␣	onLoad=	␣	`alert(1)`	␣	ϵ	␣	`</script>">`
`"><script>`	ϵ	ϵ	ϵ	ϵ	ϵ	`alert(1)`	ϵ	ϵ	ϵ	`</script>">`
`"><script>`	ϵ	ϵ	ϵ	onError=	ϵ	`alert(1)`	ϵ	ϵ	ϵ	`</script>">`

This SUT also does not perform any filtering of the input parameter and the page response will include the following HTML expression after the first recommended test from the table above is reflected:

```
<input type="text" value=""><script>   alert(1)  </script>">">
```

This vector, as well as the other recommended 5-way tests, triggers the XSS vulnerability because of the embedded inducing combination. First the `value` field and the `input` tag are terminated and then a new `script` environment with the payload is created. Upon rendering the payload inside the script environment is then executed.

SUT 3. In the initial test execution 228 test vectors triggered the XSS vulnerability. Based on these results, BEN recommended 10 tests using the found 4-way suspicious combinations. All of these tests triggered the vulnerability. We also instructed BEN to look for 2-way and 3-way suspicious combinations but none were found. This means that the reported 4-way inducing combinations are truly 4-way and not lower-strength inducing combinations embedded inside higher-strength combinations. The recommended tests and the inducing combinations are displayed in Table 3.

As this SUT encodes angle brackets a page response contains the following HTML part after the first recommended test is reflected:

Table 3. Recommended tests with embedded 4-way inducing combinations

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
<<script>	␣	">	␣	onError=	␣	alert(1)	␣	'>	␣	\>
<<script>	␣	">	␣	onError=	␣	alert(1)	␣	')	␣	\>
<<script>	␣	">	␣	onError=	␣	'alert(1)'	␣	'>	␣	\>
<<script>	␣	">	␣	onError=	␣	'alert(1)'	␣	')	␣	\>
<script	␣	">	␣	onError=	␣	alert(1)	␣	')	␣	\>
<script	␣	">	␣	onError=	␣	alert(1)	␣	<	␣	\>
<script	␣	">	␣	onError=	␣	'alert(1)'	␣	')	␣	\>
<script	␣	">	␣	onError=	␣	'alert(1)'	␣	<	␣	\>
<script>	␣	">	␣	onError=	␣	alert(1)	␣	<	␣	\>
<<script>	␣	">	␣	onError=	␣	alert(1)	␣	<	␣	\>

```
<img src="&lt;&lt;script&gt;"&gt; onError= alert(1) '&gt; \&gt;">
```

This vector succeeds in triggering the vulnerability because it contains an inducing combination which first defines the `src` attribute as `"&lt;&lt;script&gt; "` which of course is not a valid image resource. This causes the `onError` handler to be called which activates the payload, in this case `alert(1)`.

SUT 4. The initial test execution showed 280 vectors to trigger the XSS vulnerability. As all ten tests recommended by BEN for 4-way suspicious combinations triggered the vulnerability five inducing combinations were found. As for SUT 3, we also instructed BEN to look for 2-way and 3-way suspicious combinations but none were found, meaning that the inducing 4-way combinations are minimal inducing combinations. The recommended tests can be found in Table 4.

Table 4. Recommended tests with embedded 4-way inducing combinations

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
"><script>	␣	';	␣	onError=	␣	alert(1)	␣	'>	␣	\>
"><script>	␣	'>	␣	onError=	␣	alert(1)	␣	'>	␣	\>
"><script>	␣	'>	␣	onError=	␣	alert(1)	␣	')	␣	\>
"><script>	␣	ϵ	␣	onError=	␣	alert(1)	␣	'>	␣	\>
"><script>	␣	ϵ	␣	onError=	␣	alert(1)	␣	')	␣	\>
"><script>	␣	'>>	␣	onError=	␣	alert(1)	␣	')	␣	\>
"><script>	␣	';	␣	onError=	␣	src="invalid"	␣	'>	␣	\>
"><script>	␣	'>	␣	onError=	␣	src="invalid"	␣	'>	␣	\>
"><script>	␣	'>	␣	onError=	␣	src="invalid"	␣	')	␣	\>
"><script>	␣	ϵ	␣	onError=	␣	src="invalid"	␣	'>	␣	\>

To illustrate how the inducing combinations trigger the vulnerability consider the example below which shows the first recommended test vector from the above table reflected in the page response after all angle brackets and single quotes have been encoded by the SUT.

```
<img src=""&gt;&lt;script&gt; &#39;; onError= alert(1) &#39;&gt; \&gt;">
```

Here the vector succeeds in triggering the vulnerability because it first closes the `src` attribute leaving it empty. Since the empty string is not a valid resource the `onError` handler is called which in turn calls the `alert(1)` statement.

8 Conclusion and Future Work

In this paper we have presented a combinatorial approach to analyzing XSS vulnerabilities in web applications. Our approach is based on the notion of XSS-inducing combinations. An XSS-inducing combination is a combination of input parameter values whose appearance in a test vector would definitely result in a successful triggering of an XSS vulnerability at runtime when executed against the SUT. Identification of XSS-inducing combinations helps to better understand the root cause of an XSS vulnerability and provides insights about how to fix a flawed sanitization function. Our approach is developed based on our earlier work on applying combinatorial methods to security testing. In particular, our approach consists of a refinement of a combinatorial model of XSS attack vectors and a modification of a combinatorial testing-based fault localization method that are developed in our earlier works. We have reported an experiment in which our approach is applied to four sanitization functions from WAVSEP. The experimental results show that our approach can effectively identify XSS-inducing combinations and that these combinations provide significant insights about the inner working of these sanitization functions.

We plan to continue our work in the following three directions. First, we plan to conduct additional experiments for a more thorough evaluation of our approach. In particular, we plan to apply our approach to more sanitization functions that are found in real-life web applications. Second, we plan to apply our approach to other types of vulnerabilities, e.g., SQL injections. We believe that the principles embodied in our approach are general, i.e. not limited to XSS vulnerabilities. Finally, we plan to build a software tool that automates our approach with the goal to make our approach accessible to web application developers.

Acknowledgments. This work has been funded by the Austrian Research Promotion Agency (FFG) under grant 851205 and the Austrian COMET Program (FFG).

References

1. Argyros, G., Stais, I., Kiayias, A., Keromytis, A.G.: Back in black: towards formal, black box analysis of sanitizers and filters. In: Proceedings of the 37th IEEE Symposium on Security and Privacy (2016)
2. Bozic, J., Garn, B., Kapsalis, I., Simos, D., Winkler, S., Wotawa, F.: Attack pattern-based combinatorial testing with constraints for web security testing. In: Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, pp. 207–212 (2015)

3. Bozic, J., Garn, B., Simos, D.E., Wotawa, F.: Evaluation of the IPO-family algorithms for test case generation in web security testing. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10 (2015)
4. Brcic, M., Kalpic, D.: Combinatorial testing in software projects. In: Proceedings of the 35th International Convention, MIPRO, 2012 , pp. 1508–1513 (2012)
5. Duchene, F., Groz, R., Rawat, S., Richier, J.L.: XSS vulnerability detection using model inference assisted evolutionary fuzzing. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST 2012, pp. 815–817. IEEE Computer Society, Washington (2012)
6. Duchene, F., Rawat, S., Richier, J.L., Groz, R.: KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In: CODASPY. ACM (2014)
7. Garn, B., Kapsalis, I., Simos, D., Winkler, S.: 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, pp. 16–21. ACM (2014)
8. Ghandehari, L.S., Lei, Y., Kung, D., Kacker, R., Kuhn, R.: Fault localization based on failure-inducing combinations. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 168–177. IEEE (2013)
9. Ghandehari, L.S.G., Lei, Y., Xie, T., Kuhn, R., Kacker, R.: Identifying failure-inducing combinations in a combinatorial test set. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 370–379. IEEE (2012)
10. Grindal, M., Offutt, J.: Input parameter modeling for combination strategies. In: Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering SE 2007, pp. 255–260. ACTA Press, Anaheim (2007)
11. Hagar, J.D., Wissink, T.L., Kuhn, D., Kacker, R.N.: Introducing combinatorial testing in a large organization. Computer **48**(4), 64–72 (2015)
12. Hydara, I., Sultan, A.B.M., Zulzalil, H., Admodisastro, N.: Current state of research on cross-site scripting (XSS) a systematic literature review. Inf. Softw. Technol. **58**, 170–186 (2015)
13. Kuhn, D.R., Okun, V.: Pseudo-exhaustive testing for software. In: 30th Annual IEEE/NASA Software Engineering Workshop, SEW 2006, pp. 153–158. IEEE (2006)
14. Kuhn, D., Kacker, R., Lei, Y.: Introduction to Combinatorial Testing. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis (2013)
15. van der Loo, F.: Comparison of penetration testing tools for web applications. Master's thesis, University of Radboud, Netherlands (2011)
16. Mohammadi, M., Chu, B., Lipford, H.R., Murphy-Hill, E.: Automatic web security unit testing: XSS vulnerability detection. In: Proceedings of the 11th International Workshop on Automation of Software Test, AST 2016, pp. 78–84. ACM, New York (2016)
17. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 11: 1–11: 29 (2011)
18. Shi, L., Nie, C., Xu, B.: A software debugging method based on pairwise testing. In: Sunderam, V.S., Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3516, pp. 1088–1091. Springer, Heidelberg (2005). doi:10.1007/11428862_179

19. Sudhodanan, A., Armando, A., Carbone, R., Compagna, L.: Attack patterns for black-box security testing of multi-party web applications. In: Proceedings of the Network and Distributed system Security Symposium (NDSS) (2016)
20. Tripp, O., Weisman, O., Guy, L.: 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, pp. 347–357. ACM, New York (2013)
21. Wang, Z., Xu, B., Chen, L., Xu, L.: Adaptive interaction fault location based on combinatorial testing. In: 2010 10th International Conference on Quality Software (QSIC), pp. 495–502. IEEE (2010)
22. Williams, J., Wichers, D.: OWASP Top 10 2013 (2013). https://www.owasp.org/index.php/Top_10_2013
23. Yu, L., Lei, Y., Kacker, R., Kuhn, D.: Acts: a combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 370–375 (2013)
24. Zhang, Z., Zhang, J.: Characterizing failure-causing parameter interactions by adaptive testing. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 331–341. ACM (2011)