2015 IEEE International Conference on Software Quality, Reliability and Security

Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing

Josip Bozic*, Bernhard Garn†, Ioannis Kapsalis†, Dimitris E. Simos†, Severin Winkler‡ and Franz Wotawa*
*Institute for Software Technology, Graz University of Technology, A-8010 Graz, Austria
Email: jbozic@ist.tugraz.at, wotawa@ist.tugraz.at
†SBA Research, A-1040 Vienna, Austria
Email: ikapsalis@sba-research.org, bgarn@sba-research.org, dsimos@sba-research.org, swinkler@sba-research.org

***Abstract*—Security testing of web applications remains a major problem of software engineering. In order to reveal vulnerabilities, manual and automatic testing approaches use different strategies for detection of certain kinds of inputs that might lead to a security breach. In this paper we compared a state-of-the-art manual testing tool with an automated one that is based on model-based testing. The first tool requires user input from the tester whereas the second one reduces the necessary amount of manual manipulation. Both approaches depend on the corresponding test case generation technique and its produced inputs are executed against the system under test (SUT). For this case we enhance a novel technique, which combines a combinatorial testing technique for input generation and a model-based technique for test execution. In this work the input parameter modelling is improved by adding constraints to generate more comprehensive and sophisticated testing inputs. The evaluated results indicate that both techniques succeed in detecting security leaks in web applications with different results, depending on the background logic of the testing approach. Last but not least, we claim that attack pattern-based combinatorial testing with constraints can be an alternative method for web application security testing, especially when we compare our method to other test generation techniques like fuzz testing.**

Keywords*—Combinatorial testing, model-based testing, web security testing, attack patterns, injection attacks.

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. Cross-site scripting (XSS) constitutes one of the most serious vulnerabilities according to the Open Web Application Security Project (OWASP). We focus on exploiting XSS vulnerabilities and in particular we distinguish between two different types of XSS, namely reflected XSS (RXSS) and stored XSS (SXSS).

The authors from [1] depict an attack model in form of a UML state machine and describe a testing tool that uses libraries of attack inputs for test case execution. The biggest difference to our work presented here is the proposed test generation technique and also the technical implementation of the approach. We also put great emphasis on the modeling and detection of XSS. A comparison of several penetration testing tools is given by van der Loo in [2]. In this work, several web applications are tested with commercial as well as open source penetration testing tools. However, when testing for XSS vulnerabilities our approach achieves much more positive results. Attack grammars for generation of XSS attack vectors via learning and fuzzing approaches were employed in [3] and [4], respectively, where some based on combinatorial modelling have been presented in [5], [6].

In this paper, we propose an enhanced automated penetration testing technique, whereby the presented modelling builds upon the work from [5]. Furthermore, the paper includes a detailed case study for testing various SUTs against our automated testing method as well as manual testing approaches and presents a detailed evaluation and comparison. We also draw useful conclusions that further strengthen the applicability of attack pattern-based combinatorial testing to security testing, building on the comparison of the test results of the testing methods given in [5], [6]. Most important, we present an extensive comparison in terms of (total) vulnerabilities found by our generated test suites versus ones that are being produced mostly as a result from fuzzers and our findings indicate that our combinatorial test generation method rises as an alternative approach for web security testing. The main *contributions* of this paper are summarized as follows:

- Remodelled XSS attack grammar which also includes a set of constraints. The test inputs (XSS attack vectors) are generated in an automated way;
- Revision of the attack pattern-based combinatorial testing technique with respect to the new grammar;
- Detailed evaluation of a case study for web security testing, including automated and manual test execution methods;
- Extensive comparison between the test generation component of the attack pattern-based combinatorial testing technique with various fuzzers.

The paper is structured as follows: In Section I we elaborate on a detailed explanation of combinatorial testing as well as its potential contribution to security testing. Afterwards, Sections II and III describe the goals of our case study and the penetration testing execution methods, namely the attack pattern-based approach and manual penetration testing frameworks. Then Section IV discusses the testing results of our automated and manual approach against several web applications and finally, Section V concludes the work.

The work of the fourth author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.
Authors are listed in alphabetical order.

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.38

I. Combinatorial Security Testing

An overview of the combinatorial test design process can be found in [7] and in [5], [6] for a more related usage to security testing. In particular, we used the ACTS combinatorial test generation tool [8] for automated test generation of inputs and subsequently the attack pattern-based methodology given in Section III for test execution.

Combinatorial Grammar for XSS Attack Vectors: Revisited. In this section, we consider a general structure for XSS attack vectors (test inputs) where each one of them is comprised of 11 discrete parameters (types) discussed in detail below. This structure builds upon a combinatorial grammar given in [5] by modelling white spaces and executable JavaScript that can appear in an XSS attack vector but also extends the one given in [6] by adding constraints between the different parameter values. To this point, we would like to note that we follow the terminology used in security testing when we are referring to XSS attack vectors as test inputs, as this is described for example in [9], [3]. A fragment of our combinatorial grammar, denoted by G, is presented below in BNF form so as to be able to generate possible parameter values through ACTS, where inside the parentheses in the parameters we list the full range of values we have taken into account in our implementation.

```
JSO(15)::=␣<script>␣|␣<img␣|␣''><script>␣|␣...
WS1(3)::=␣tab␣|␣space␣|␣empty
INT(14)::=␣\";␣|␣'>␣|␣">>␣|␣...
WS2(3)::=␣tab␣|␣space␣|␣empty
EVH(3)::=␣onLoad(␣|␣onMouseOver(␣|␣onError(
WS3(3)::=␣tab␣|␣space␣|␣empty
PAY(23)::=␣alert('XSS')␣|␣SRC="javascript:alert('1');">␣|␣
     HREF="http://ha.ckers.org/xss.js">␣|␣...
WS4(3)::=␣tab␣|␣space␣|␣empty
PAS(11)::=␣')␣|␣;//␣|␣'>␣|␣...
WS5(3)::=␣tab␣|␣space␣|␣empty
JSE(9)::=␣</script>␣|␣\>␣|␣'\>␣|␣...
```

BNF Grammar for XSS Attack Vectors

Note that, publicly available resources for XSS vectors are in high demand in the (industrial) testing community, see for example [9]. In our attack grammar the given parameter values are just a fragment of possible options. If the designer would like to increase the number of test inputs in a test suite, one possibility is the addition of new parameter values in the given BNF for XSS attack vectors. As a result, the generated combinatorial object will also grow. Based on the presented attack grammar and incorporating expert knowledge we will build upon the following form of an XSS attack vector (AV): **(JSO,WS1,INT,WS2,EVH,WS3,PAY,WS4,PAS,WS5,JSE)**.

The expert knowledge was essential in the design of the AV, where our goal is to produce valid JavaScript code when this is injected into SUT parameters. The description of the types in the previous AV has briefly been mentioned in [6], however we include it also here, in more detail, for the sake of completeness:

- The **JSO** (JavaScript Opening Tags) type represents tags that open a JavaScript code block. They also contain values that use common techniques to bypass certain XSS filters, like <script> or <img.
- The **WS** (white space) type family represents the white space but also variations of it like the tab character in order to circumvent certain filters.
- The **INT** (input termination) type represents values that terminate the original valid tags (HTML or others) in order to be able to insert and execute the payload, like "'> or ">.
- The **EVH** (event handler) type contains values for JavaScript event handlers. The usage of JavaScript event handlers, like onLoad(or onError(, is a common approach to bypass XSS filters that filter out the typical JavaScript opening tags like <script> or filters that remove brackets (especially < and >).
- The **PAY** (payload) type contains executable JavaScript like alert("XSS") or ONLOAD=alert('XSS')>. This type contains different types of executable JavaScript in order to bypass certain XSS filters.
- The **PAS** (payload suffix) type contains different values that should terminate the executable JavaScript payload (PAY parameter). The PAS is necessary to produce valid JavaScript code that is interpreted by a browser like ') or '>.
- The **JSE** (JavaScript end tag) type contains different forms of JavaScript end tags in order to produce valid JavaScript code like </script> or >.

The combinatorial test design process for XSS attack vector generation, has been explained in detail in [6], [10].

Adding Constraints. In this section we consider adding constraints to our XSS attack grammar. The motivation for this reason rises from the fact that in real-world systems adding constraints may produce test suites with better quality and also considerably reduce the input space. This approach for combinatorial testing has been followed for example in [11]. We would like to note that although attack grammars for XSS attack vectors have been given in [5], [6], [3], [4], this is the first time that sets of constraints are imposed on such attack grammars in terms of combinatorial modelling and our approach for revisiting the notion of *combinatorial security testing* is novel in that sense. We present below the full set of constraints for our grammar using the constraint tool from ACTS where the symbol => means an implication and the symbol || an OR statement. We will denote this grammar with G_c to distinguish it from G when constraints are enforced.

```
(JSO=1) => (JSE=1)
(JSO=2) => (JSE=2)
(JSO=3) => (JSE=3)
(JSO=4) => (JSE=2 || JSE=4)
(JSO=5) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=6) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=7) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=9) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=10) => (JSE=9)
(JSO=11) => (JSE=2 || JSE=3 || JSE=4)
(JSO=12) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=13) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(JSO=14) => (JSE=2 || JSE=3 || JSE=4)
(JSO=15) => (JSE=2 || JSE=3 || JSE=4)
(EVH=1) => (PAY=12 || PAY=14 || PAY=17 || PAY=18 || PAY=19)
(EVH=2) => (PAY=13 || PAY=14 || PAY=17 || PAY=18 || PAY=19)
(EVH=3) => (PAY=12 || PAY=13 || PAY=17 || PAY=18 || PAY=19)
(INT=2) => (PAS=10 || PAS=11)
(INT=3) => (PAS=10 || PAS=11)
(INT=4) => (PAS=10 || PAS=11)
(INT=5) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=6) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=7) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=8) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=9) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=10) => (PAS=6 || PAS=7 || PAS=8 || PAS=9)
(INT=11) => (PAS=10 || PAS=11)
(WS1=WS2 && WS2=WS3 && WS3=WS4 && WS4=WS5)
```

Constraints for XSS Attack Grammar

We remark that in our implementation, we used a translation layer from the integer values presented previously to actual values per derivation type, in a similar manner to the ones presented for G. The rationale for the used constraints by incorporating the expert knowledge is as follows:

- Constraints of the form `(JSO=x) => (JSE=y)` for appropriate values x and y ensure coupling of the outermost JavaScript opening and closing tags. An alignment between these two types is necessary to produce valid JavaScript code.
- Constraints of the form `(EVH=x) => (PAY=y)` for appropriate values x and y offer the possibility to use only those payload values, which are, derived from expert knowledge, most likely to succeed when coupled with the respective event handler.
- Constraints of the form `(INT=x) => (PAS=y)` for appropriate values x and y further express necessary correlations/conditions between the two stages to be able to ultimately execute the payload type, which an experienced tester would intuitively use when performing manual penetration testing.
- Constraints between the white space types mainly serve to reduce the overall size of the test suites.

Combinatorial Metrics for Security Testing. There has been a great need for metrics, e.g. how to measure the efficiency of testing experiments, in software security the latest years. Many different notions of coverage criteria used in traditional software testing such as branch coverage and statement coverage have been adopted by security researchers [12]. In this work, to avoid confusion with the term of combinatorial coverage[1], we will instantiate a metric used in combinatorial testing to measure the efficiency of our test suites when these are executed versus SUTs that are web applications and we target to exploit XSS vulnerabilities. In particular, in [7] the success rate of test suites when these are executed versus various SUTs is defined as, $P :=$ set of t-way combinations in passing tests. In the context of automated security testing for web applications (c.f. [12]), we define as the *exploitation rate* of an SUT, denoted by ER, the proportion of XSS attack vectors that were successful, e.g. the ones that exploit an XSS vulnerability, per given test suite and SUT:

$$\mathrm{ER} = \frac{\#\text{ Attack vectors that exploit an XSS vulnerability}}{\text{Total number of attack vectors per test suite and SUT}}$$

Even though from a security testing point of view, it might be sufficient for at least one XSS attack vector to be successful, for combinatorial testing it is of utter importance to estimate the quality of the generated test suites. Reverse engineering the structure of successful vectors, which implies remodelling the attack grammar, we can hope to achieve better results in terms of exploitation rate but also to find vulnerabilities in SUTs that we were unable to penetrate before in [5].

II. Case Study

The SUTs used in our case study comprised a set of web applications that are included in the Open Web Application Security Project (OWASP) Broken Application Project[2] and in the Exploit Database Project[3]. Some of these applications, i.e. Webgoat, Mutillidae and DVWA, were already tested in [5] and [6] but we added WebGoat[4], Gruyere[5] and Bitweaver[6] to the list of SUTs. Also, Mutillidae and DVWA comprise several difficulty levels, every one of them activating additional built-in filtering mechanisms against submitted inputs. All of these programs are web applications that were deployed locally and comprise a corresponding database.

One of the goals of our case study is to investigate how our attack pattern-based combinatorial testing method behaves in relation to manual based approaches. This in effect, will indicate whether we can further strengthen the applicability of our method to web security testing. Secondly, on the same level of importance, we want to explore whether combinatorial testing can be an alternative method when compared to fuzz testing for usage in web security testing methodologies. In particular, we want to compare the exploitation rate between test suites that have been generated with combinatorial testing versus ones that are produced with various fuzzers (c.f. XSS repos) focused on triggering XSS exploits (higher exploitation rate is better). The SUTs were tested with different sets of test suites, produced by the ACTS tool and based on the combinatorial grammar given in Section I. The difference between the two sets of test suites rely on imposing various constraints on the different types of the attack grammar. This issue was pointed to us by a reviewer of [5] and thus is intriguing to investigate whether in our experiments we will confirm the findings of [11], which states that imposing constraints on real-world applications makes higher strength combinatorial interaction testing feasible.

III. Penetration Testing Execution Methods

In this section we provide details about the two penetration testing execution methods, automated and manual ones, we have used in our case study. We give a description of their test oracles as their functionality has been described already in past works. Both methods can be applied to security testing, 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.

Attack Pattern-Based Testing: Revisited. This method is taken from [13] and elaborated in more detail in [14], [15]. Its main characteristics are the usage of a model of an attack, which itself is initially manually modelled in form of a UML statechart with help of the open source tool-kit YAKINDU Statechart Tools. In our case, the attack model consists of attack steps for detection of both types of XSS. Because the expected outcome might be hard to predict, the implemented test oracle relies on encountering an unexpected HTML element like *<script>*, *<img>* etc. that serves as an indicator for unwanted behavior caused by the input. After a

[1]Refers to the covered t-way tuples of given k parameters of an SUT in a mixed-level covering array.

[2]https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project

[3]http://www.exploit-db.com/

[4]https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[5]http://google-gruyere.appspot.com/

[6]http://www.bitweaver.org/

HTTP request is sent with a malicious input value against some element of the web application, its corresponding response is parsed by a HTML parser. If an expected outcome has been generated during the process, a vulnerability is considered to be detected.

Manual Penetration Testing Tools. The Burp Suite[7] is an integrated platform for performing security testing of web applications. It is widely used by security professionals since it allows them to perform many penetration testing tasks [6]. The oracle used within Burp Suite was enabled by using the "Search responses for payload strings" configuration option within the intruder. This option flags all results where the payloads were exactly reflected to the response. The rationale behind this decision is that if the vector was not blocked or potential dangerous characters were not stripped out, we assume an XSS vulnerability was triggered. Additionally, for the cases where we tested for stored XSS, we enable the option "Follow redirections:On-site only" in order to catch the redirections triggered from the injected vectors that we managed to stored on the server side.

IV. Evaluation

As described in Section I we formulated a more complex XSS grammar for the input model in ACTS and generated inputs for combinatorial interaction strengths $t \in \{2, 3, 4\}$ twice. The first dataset consists of inputs that were created without setting any constraints on the model and comprises different test suites depending on the strength while the second one comprises of analogue test suites, which were generated according to constraints introduced in Section I. We tested all mentioned applications against both attack inputs and displayed the responsive results. In the first case, for interaction strength $t = 2$ the combinatorial tool generated 345 inputs and respectively 4875 and 53706 attack strings for $t = 3$ and $t = 4$. Because constraints put a limitation on the data structure, a remarkably smaller amount of strings were created for the second dataset, namely 250, 1794 and 8761 inputs. Both sets were used in the attack pattern-based approach and in Burp Suite so a comparison could be made according to the results from both cases. In order to draw a meaningful comparison, we tested the same parts of a SUT, which were input fields with textual and password values or textarea tags but we also attached attack strings to the URL paths without any variable binding.

Exploitation Rate of SUTs. In this section we investigate how the exploitation rate of the different SUTs we considered in our case study scales when the combinatorial interaction strength increases, for given difficulty level and input field in each one of the SUTs per penetration testing tool. The evaluation results are depicted in Table I for the XSS combinatorial grammar we have modelled (with and without constraints on the derivation types) and when the generated vectors where executed using our attack pattern-based testing method and also with a manual penetration testing framework (Burp suite). In particular, in Table I we give information about the combinatorial interaction strength (Str.), the SUT (App), the input parameter ID (inp_ID), type of vulnerability (VT), eventually the difficulty level (DL), the exploitation rate (the number of positive inputs divided by the total number of tested vectors) and its respective percentage. Further, the table gives the corresponding results for constrained values, where in only a few cases in Gruyere some test runs were not completed due to its unexpected behavior (denoted by "N/A" in the table).

In the majority of the input fields of the SUTs we considered in our case study we confirm the fundamental rule of combinatorial testing; testing with higher interaction strengths is likely to reveal more errors (see for example WebGoat, BodgeIt in Table I). In our context, we interpret and confirm this rule in terms of exploitation rate, e.g. increasing the interaction strength implies higher exploitation rates of web applications when these are tested for XSS vulnerabilities.

Evaluation of Combinatorial Grammars. In almost all of our experiments, a better exploitation rate was achieved by applying constraints upon input generation, which leads to the conclusion that even better results might be achieved by setting even more constraints but also testing with greater combinatorial interaction strengths. For both input sets we got a somehow higher rate with increasing t. In all test runs when testing with attack vectors generated with constraints, we either had a significant increase of the exploitation rate or exactly the same was achieved. In detail, in some test runs the improvement in the exploitation rate is up to 7 times greater than the exploitation rate achieved with the vectors generated by the combinatorial grammar without constraints. We believe that the reason the obtained results when constraints were applied are better (from the ones without constraints) because we generated a test suite with better quality, i.e. by excluding many low quality attack vectors.

Comparison of Fuzzers and Combinatorial Testing. In this section, we change the focus of our evaluation and now we aim to investigate how our test suites generated by combinatorial testing compare to fuzzers. We have collected a number of such test suites (produced by fuzzers), that are publicly available and executed them against the same SUTs using both automated and manual test case execution methods. We compare the exploitation rate for the vectors produced with combinatorial and fuzz testing within the same test execution method (when these are tested against the same web application), in order to draw more accurate conclusions. In the evaluation results presented in Table II we take into account the best exploitation rate achieved with combinatorial testing for the same test run from Table I, denoted by best CT % ER. When comparing the exploitation rate of fuzzers and combinatorial testing in Table II, as before, we give information for the SUT (App), the input parameter ID (inp_ID), type of vulnerability (VT) for our attack pattern-based testing method and also with the manual testing tool, Burp. In particular, we have considered the following resources:

1) OWASP XSS Filter Evasion Cheat Sheet[8] with 113 vectors.
2) Attack and Discovery Pattern Database for Application Fuzz Testing[9] (rsnake) with 76 vectors.
3) HTML5 Security Cheat Sheet[10] with 170 vectors.

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

[8] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[9] https://code.google.com/p/fuzzdb

[10] https://html5sec.org/

TABLE I. EVALUATION RESULTS PER SUT FOR GIVEN DIFFICULTY LEVEL AND INPUT FIELD WITH INCREASING STRENGTH.

SUT parameters				Str.	Attack Pattern-Based Testing				Manual Testing (Burp Suite)			
					G		G_c		G		G_c	
App	DL	VT	inp_id		ER	% ER	ER	% ER	ER	% ER	ER	% ER
Webgoat	0	RXSS	2	2	198/345	57.39	145/250	58.00	177/345	51.30	156/250	62.40
Webgoat	0	RXSS	2	3	2842/4875	58.3	1073/1794	59.81	4240/4875	86.97	1672/1794	93.20
Webgoat	0	RXSS	2	4	31441/53706	58.54	5366/8761	61.26	47249/53706	87.98	8586/8761	98.00
BodgeIt	0	RXSS	1	2	175/345	50.72	145/250	58.00	315/345	91.30	250/250	100.00
BodgeIt	0	RXSS	1	3	2518/4875	51.64	1073/1794	59.81	4445/4875	91.18	1794/1794	100.00
BodgeIt	0	RXSS	1	4	31441/53706	58.54	5366/8761	61.25	49012/53706	91.26	8761/8761	100.00
BodgeIt	0	RXSS	2	2	107/345	31.01	97/250	38.80	9/345	2.61	40/250	16.00
BodgeIt	0	RXSS	2	3	1564/4875	32.08	737/1794	41.08	117/4875	2.40	264/1794	14.72
BodgeIt	0	RXSS	2	4	20926/53706	38.96	3918/8761	44.72	1279/53706	2.38	1379/8761	15.74
BodgeIt	0	SXSS	3	2	31/345	8.99	42/250	16.80	57/345	16.52	75/250	30.00
BodgeIt	0	SXSS	3	3	561/4875	11.51	294/1794	16.39	831/4875	17.05	524/1794	29.21
BodgeIt	0	SXSS	3	4	6052/53706	11.27	1481/8761	16.90	8996/53706	16.75	2531/8761	28.89
BodgeIt	0	SXSS	4	2	308/345	89.28	175/250	70.00	0/345	0.00	0/250	0.00
BodgeIt	0	SXSS	4	3	4434/4875	90.95	1264/1794	70.46	0/4875	0.00	0/1794	0.00
BodgeIt	0	SXSS	4	4	42899/53706	79.88	6172/8761	70.45	0/53706	0.00	1/8761	0.01
DVWA	0	RXSS	1	2	175/345	50.72	128/250	51.20	315/345	91.30	250/250	100.00
DVWA	0	RXSS	1	3	2517/4875	51.63	954/1794	53.18	4445/4875	91.18	1794/1794	100.00
DVWA	0	RXSS	1	4	27864/53706	51.88	4755/8761	54.28	49012/53706	91.26	8761/8761	100.00
DVWA	0	SXSS	2	2	104/345	30.14	98/250	39.20	150/345	43.48	138/250	55.20
DVWA	0	SXSS	2	3	1364/4875	27.98	565/1794	31.50	2149/4875	44.08	996/1794	55.52
DVWA	0	SXSS	2	4	13862/53706	25.81	2581/8761	29.46	23402/53706	43.57	4739/8761	54.09
DVWA	1	RXSS	1	2	106/345	30.72	80/250	32.00	210/345	60.87	170/250	68.00
DVWA	1	RXSS	1	3	1547/4875	31.73	613/1794	34.17	2966/4875	60.84	1219/1794	67.95
DVWA	1	RXSS	1	4	17172/53706	31.97	3285/8761	37.50	32724/53706	60.93	6223/8761	71.03
DVWA	1	SXSS	2	2	0/345	0	0/250	0.00	0/345	0.00	0/250	0.00
DVWA	1	SXSS	2	3	0/4875	0	0/1794	0.00	0/4875	0.00	0/1794	0.00
DVWA	1	SXSS	2	4	0/53706	0	0/8761	0.00	2/53706	0.00	6/8761	0.07
Gruyere	0	RXSS	1	2	122/345	35.36	89/250	35.60	315/345	91.30	250/250	100.00
Gruyere	0	RXSS	1	3	1744/4875	35.77	671/1794	37.40	4445/4875	91.18	1794/1794	100.00
Gruyere	0	RXSS	1	4	19382/53706	36.09	3303/8761	37.70	N/A	N/A	N/A	N/A
Gruyere	0	SXSS	2	2	23/345	6.67	17/250	6.80	50/345	14.49	42/250	16.80
Gruyere	0	SXSS	2	3	326/4875	6.69	118/1794	6.58	629/4875	12.90	256/1794	14.27
Gruyere	0	SXSS	2	4	3576/53706	6.66	456/8761	5.20	N/A	N/A	N/A	N/A
Mutillidae	0	RXSS	1	2	111/345	32.17	116/250	46.40	345/345	100.00	250/250	100.00
Mutillidae	0	RXSS	1	3	1580/4875	32.41	836/1794	46.60	4875/4875	100.00	1794/1794	100.00
Mutillidae	0	RXSS	1	4	17344/53706	32.29	1833/8761	20.92	53706/53706	100.00	8761/8761	100.00
Mutillidae	0	RXSS	2	2	158/345	45.8	161/250	64.40	63/345	18.26	83/250	33.20
Mutillidae	0	RXSS	2	3	2304/4875	47.26	1153/1794	64.27	921/4875	18.89	581/1794	32.39
Mutillidae	0	RXSS	2	4	25199/53706	46.92	5521/8761	63.02	9803/53706	18.25	2812/8761	32.10
Mutillidae	0	RXSS	3	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
Mutillidae	0	RXSS	3	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
Mutillidae	0	RXSS	3	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00
Mutillidae	1	RXSS	1	2	111/345	32.17	116/250	46.40	345/345	100.00	250/250	100.00
Mutillidae	1	RXSS	1	3	1580/4875	32.41	836/1794	46.60	4875/4875	100.00	1794/1794	100.00
Mutillidae	1	RXSS	1	4	17344/53706	32.29	1833/8761	20.92	53706/53706	100.00	8761/8761	100.00
Mutillidae	1	RXSS	2	2	158/345	45.8	161/250	64.40	63/345	18.26	83/250	33.20
Mutillidae	1	RXSS	2	3	2304/4875	47.26	1154/1794	64.33	921/4875	18.89	581/1794	32.39
Mutillidae	1	RXSS	2	4	25199/53706	46.92	5521/8761	63.02	9803/53706	18.25	2812/8761	32.10
Mutillidae	1	RXSS	3	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
Mutillidae	1	RXSS	3	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
Mutillidae	1	RXSS	3	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00
Bitweaver	0	RXSS	1	2	0/345	0	0/250	0.00	72/345	20.87	84/250	33.60
Bitweaver	0	RXSS	1	3	0/4875	0	0/1794	0.00	1269/4875	26.03	588/1794	32.78
Bitweaver	0	RXSS	1	4	0/53706	0	0/8761	0.00	14225/53706	26.49	2904/8761	33.15
Bitweaver	0	RXSS	2	2	0/345	0	0/250	0.00	72/345	20.87	84/250	33.60
Bitweaver	0	RXSS	2	3	0/4875	0	0/1794	0.00	1269/4875	26.03	588/1794	32.78
Bitweaver	0	RXSS	2	4	0/53706	0	0/8761	0.00	14225/53706	26.49	2904/8761	33.15
Bitweaver	0	RXSS	3	2	198/345	57.39	145/250	58.00	345/345	100.00	250/250	100.00
Bitweaver	0	RXSS	3	3	2842/4875	58.69	1073/1794	59.81	4875/4875	100.00	1794/1794	100.00
Bitweaver	0	RXSS	3	4	31441/53706	58.54	5366/8761	61.25	53706/53706	100.00	8761/8761	100.00
Bitweaver	0	RXSS	4	2	0/345	0	0/250	0.00	345/345	100.00	250/250	100.00
Bitweaver	0	RXSS	4	3	0/4875	0	0/1794	0.00	4875/4875	100.00	1794/1794	100.00
Bitweaver	0	RXSS	4	4	0/53706	0	0/8761	0.00	53706/53706	100.00	8761/8761	100.00

TABLE II. EVALUATION RESULTS FOR FUZZERS AND COMBINATORIAL TESTING PER SUT FOR GIVEN DIFFICULTY LEVEL AND INPUT FIELD.

SUT parameters				Attack Pattern-Based Testing					Manual Testing (Burp Suite)				
				Fuzzers				Best CT	Fuzzers				Best CT
App	DL	VT	inp_id	OWASP	Rsnake	HTML 5SEC	Xenotix	% ER	OWASP	Rsnake	HTML 5SEC	Xenotix	% ER
Mutillidae	0	RXSS	1	41.59	43.42	32.94	38.91	46.60	99.12	97.37	93.53	88.82	100.00
Mutillidae	0	RXSS	2	71.68	60.53	92.94	89.86	64.40	39.82	30.26	61.76	63.86	33.20
Mutillidae	0	RXSS	3	2.65	0.00	16.47	9.35	0.00	83.19	82.89	74.12	78.37	100.00
Mutillidae	1	RXSS	1	41.59	43.42	32.94	38.91	46.60	99.12	96.05	93.53	88.82	100.00
Mutillidae	1	RXSS	2	71.68	60.53	92.94	89.86	64.40	39.82	30.26	61.76	63.86	33.20
Mutillidae	1	RXSS	3	2.65	0.00	16.47	9.35	0.00	83.19	82.89	74.12	78.37	100.00
BodgeIt	0	RXSS	1	43.36	46.05	32.94	40.88	61.25	98.23	97.37	90.59	85.69	100.00
BodgeIt	0	RXSS	2	43.36	46.05	22.35	28.52	44.72	7.96	0.00	5.29	1.11	16.00
BodgeIt	0	SXSS	3	12.39	23.68	23.53	28.12	16.90	46.02	15.79	0.00	0.00	30.00
BodgeIt	0	SXSS	4	71.68	85.53	40.59	51.60	90.95	0.00	0.00	0.00	0.00	0.01
Gruyere	0	RXSS	1	6.19	3.95	14.12	19.23	37.70	83.19	82.89	74.12	78.37	100.00
Gruyere	0	SXSS	2	0.00	0.00	7.06	11.25	6.80	6.19	3.95	6.47	26.41	16.80
WebGoat	0	RXSS	2	43.36	46.05	33.53	41.39	61.26	39.82	57.89	42.35	76.67	98.00
DVWA	0	RXSS	1	41.59	43.42	31.76	40.88	54.28	83.19	82.89	74.12	78.37	100.00
DVWA	0	SXSS	2	39.82	31.58	30.59	27.73	39.20	92.04	93.42	84.12	68.56	55.52
DVWA	1	RXSS	1	41.59	43.42	20.59	27.99	37.50	83.19	82.89	60.00	66.93	71.03
DVWA	1	SXSS	2	0.00	0.00	0.00	0.00	0.00	0.88	0.00	0.59	2.48	0.07
Bitweaver	0	RXSS	1	0.00	0.00	0.00	0.00	0.00	26.55	40.79	8.24	26.73	33.60
Bitweaver	0	RXSS	2	0.00	0.00	0.00	0.00	0.00	26.55	40.79	8.24	26.73	33.60
Bitweaver	0	RXSS	3	43.36	46.05	32.94	41.14	59.81	99.12	96.05	93.53	88.82	100.00
Bitweaver	0	RXSS	4	0.00	0.00	0.00	0.00	0.00	99.12	96.05	93.53	88.82	100.00

4) OWASP Xenotix XSS Exploit Framework[11] where we extracted its 1530 vectors.

Comparing the exploitation rate that the different test inputs achieve against the SUTs in both tools, we can argue that the diversity of the vectors generated with combinatorial testing has achieved better results than fuzzers in some cases. In other words, the fact that we can generate test suites with different sizes (attributed to the interaction strength) for given SUT offers us a larger attack surface when compared to the one achieved with fuzz testing. Moreover, the use of constraints in the attack grammar can filter out many low quality attack vectors and this is another reason why combinatorial testing outperforms fuzz testing in some test runs. Clearly, these two features of combinatorial testing cannot be achieved with fuzz testing. We would also like to note that even in the case where combinatorial testing and fuzz testing achieve the same exploitation rate, in practice the number of actual positive inputs differs since the size of test suites generated with combinatorial testing is quite larger. To give an example, for $t = 4$ we have 8761 attack vectors when `G_C` is considered. To conclude with, it is evident from Table II, that in half of the test runs in both automated and manual test execution methods, inputs generated with combinatorial testing achieve better exploitation rates. It is safe to argue at this point that combinatorial testing can be an alternative method for test input generation, when compared to fuzz testing, applicable to web application security testing.

Comparison of Automated vs. Manual Test Execution Methods. The obtained results rely heavily on the test oracles from the tools. While testing with Burp, there have been several cases where a detection rate of 100% is noticed. This would indicate that every submitted vector was able to trigger a vulnerability. In other words, both testing procedures have produced a certain amount of false positives, which influenced the final results. The used tools were not able to detect such potential outcomes in an automatic manner. We believe that the obtained results depend more on the test execution method rather than the quality of the inputs due to different mechanisms that take place on the test oracles of the testing tools. This argument is further supported by the fact that we noticed differences on the results not only when comparing the combinatorial grammars themselves, but also when comparing the results that we obtained when testing with the vectors that were produced with fuzzers. However, we will investigate this issue further in future work.

V. Conclusion and Future Work

In this work, we revised an input grammar for combinatorial generation of test inputs and made use of constraints for another test suite. These test suites were used by both an automated and a manual testing approach in order to test several web applications for reflected and stored XSS. We would like to highlight that testing with combinatorial attack grammars with increasing interaction strength results in higher exploitation rates. Also, setting constraints inside the input model results in significantly improved attack vectors. However, in order to further improve our testing results in the attack pattern-based technique, a revisited test oracle might be taken into consideration since we witnessed some discrepancies with manual penetration testing methods. Last but not least, from the comparison of the combinatorial grammars against the ones used by fuzzers we conclude that attack pattern-based combinatorial testing can be seen as an alternative method for revealing XSS vulnerabilities in web security testing.

[11] https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework

Acknowledgements. The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and the Austrian COMET Program (FFG).

References

[1] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti, "Vera: A flexible model-based vulnerability testing tool," in *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST'13)*, 2013.

[2] F. van der Loo, "Comparison of Penetration Testing Tools for Web Applications," Master's thesis, University of Radboud, Netherlands, 2011.

[3] O. Tripp, O. Weisman, and L. 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*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 347–357.

[4] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, "XSS vulnerability detection using model inference assisted evolutionary fuzzing," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 815–817.

[5] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in *Proceedings of the 9th International Workshop on Automation of Software Test (AST)*, 2014, pp. 1–7.

[6] B. Garn, I. Kapsalis, D. E. Simos, and S. Winkler, "On the applicability of combinatorial testing to web application security testing: A case study," in *Proceedings of the 2nd International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation (JAMAICA'14)*. ACM, 2014.

[7] D. Kuhn, R. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, ser. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2013.

[8] L. Yu, Y. Lei, R. Kacker, and D. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013.

[9] A. Javed and J. Schwenk, "Towards elimination of cross-site scripting on mobile versions of web applications," in *Lecture Notes in Computer Science*, 2014, pp. 103–123.

[10] J. Bozic, B. Garn, D. Simos, and F. Wotawa, "Evaluation of the IPO-family algorithms for test case generation in web security testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–10.

[11] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, 2013, pp. 26–36.

[12] T. Dao and E. Shibayama, "Coverage criteria for automatic security testing of web applications," in *Information Systems Security*, ser. Lecture Notes in Computer Science, S. Jha and A. Mathuria, Eds. Springer Berlin Heidelberg, 2010, vol. 6503, pp. 111–124.

[13] J. Bozic and F. Wotawa, "XSS pattern for attack modeling in testing," in *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, 2013.

[14] ——, "Security testing based on attack patterns," in *Proceedings of the 5th International Workshop on Security Testing (SECTEST'14)*, 2014.

[15] A. P. Moore, R. J. Ellison, and R. Linger, "Attack Modeling for Information Security and Survivability," in *Technical Note CMU/SEI-2001-TN-001*, March 2001.