Attack Pattern-Based Combinatorial Testing

Josip Bozic
Institute for Software Technology
Graz University of Technology
A-8010 Graz, Austria
jbozic@ist.tugraz.at

Dimitris E. Simos
SBA Research
A-1040 Vienna, Austria
dsimos@sba-research.org

Franz Wotawa
Institute for Software Technology
Graz University of Technology
A-8010 Graz, Austria
wotawa@ist.tugraz.at

***Abstract*—The number of potential security threats rises with the increasing number of web applications, which cause tremendous financial and existential implications for developers and users as well. The biggest challenge for security testing is to specify and implement ways in order to detect potential vulnerabilities of the developed system in a never ending quest against new security threats but also to cover already known ones so that a program is suited against typical attack vectors. For these purposes many approaches have been developed in the area of model-based security testing in order to come up with solutions for real-world application problems. These approaches provide theoretical background as well as practical solutions for certain security issues. In this paper we partially rely on previous work but focus on the representation of attack patterns using UML state diagrams. We extend previous work in combining the attack pattern models with combinatorial testing in order to provide concrete test input, which is submitted to the system under test. With combinatorial testing we capture different combinations of inputs and thus increasing the likelihood to find weaknesses in the implementation under test that can be exploited. Besides the foundations of our approach we further report on first experiments that indicate its practical use.**

***Index Terms*—Combinatorial testing, model-based testing, security testing, attack patterns.**

I. Introduction

Covering security issues still remains a big task of the testing community in academic circles as well as industrial ones. Currently available state-of-the-art defense mechanisms of web applications against unexpected intrusion are still not implemented in all of the online systems so known security threats still represent a threat. Many manually penetration testing platforms [1][2] or automatically ones [3] for testing of known security threats are used to cover vulnerabilities against malicious actions like SQL injections, cross-site scripting (XSS) or man-in-the-middle (MITM) attacks.

On the other hand, model-based testing approaches have drawn attention in recent years. These methods use models of the SUT in order to derive test cases from them, which then are executed against the program and the output is compared to expected values in order to find out if the program behaves accordingly to its model specification. In these cases the internal structure of the SUT can be known to the tester (white-box testing) or remains utterly hidden (black-box testing).

In this paper we provide a framework for testing and detection of both reflected and stored XSS in web applications. The framework comprises two parts: a model-based testing method that is based on attack patterns, and combinatorial testing for generating the input data to be submitted to the SUT. Here the idea is to use model-based testing for generating abstract test cases, e.g., sequences of necessary interactions, and combinatorial testing for concretizing the input. In our implementation we use the combinatorial testing tool ACTS [4] for generation of input strings by specifying parameters and constraints in order to structure the inputs for the particular application domain. Once specified, these inputs are used by the attack pattern model in order to submit them to a web application.

The goal of our approach is to cover standard XSS exploitation attempts by checking whether certain parts of the SUT are vulnerable to potentially malicious scripts. For this case, the method uses a detection mechanism for recognizing unfiltered user inputs that are eventually sent back and executed at the side of the client. Also, the tester can detect what parts of the SUT are vulnerable and gets the impression how an injection is structured so further measures can be taken. The main differences to other techniques and tools ly in the generation, structure and execution of test cases.

The paper is structured as follows: Section II provides related research from both combinatorial testing related topics as well as model-based techniques for testing. Further we show in Section III a detailed explanation of combinatorial testing as well as its potential contribution to security testing. Afterwards, Sections IV and V demonstrate the attack pattern-based approach for test case execution and provide a working example in a case study. Then Section VI discusses the testing results of our approach against other web applications and finally, Section VII concludes the work.

II. Related Work

For important works in combinatorial testing that relate to our work we refer to Section III and cited references there in, while for a general treatment of the field of combinatorial testing we refer the interest reader to the recent surveys of [5], [6] and [7].

Several papers have been published that are related to model-based testing and test case execution. However, it's important to note that in this paper we do not use models of the SUT to generate test cases but as a means for carrying

Authors are listed in alphabetical order.

out attacks after inputs were previously created by the combinatorial approach.

Phillips and Swiler [8] elaborate a method by relying on attack graphs. In order to generate an attack model, an attacker profile, configuration file and attack templates are required where the last of the mentioned entities incorporates preconditions and effects for some known attacks. The approach demonstrates how an attack graph is generated where the authors start from a goal node and search backwards for templates, which edges satisfy this node and if any is found, this template is added to the attack graph. It's possible that one node gets multiple predecessor nodes and in that case additional nodes are added as long as until a start node is not reached so there is a possibility to attack the system.

The authors from [9] define attack patterns in form of UML statecharts in order to test web applications against cross-site scripting attacks. First they specify manually an adaptable XSS attack pattern model and implement Java functions, which are called from within the model once the execution takes place. The paths inside the model will lead the execution while thereby checking if the currently active transition preconditions are satisfied. If no vulnerability is triggered, then another path is taken by fulfilling further conditions so that another input may be submitted against the SUT. In the end either the final state is reached or the execution terminates because of no vulnerability detection.

A more detailed overview of XSS can be found in [10]. Also, further explanations regarding a general overview about model-based testing as well as some approaches from that area can be found in [11], [12] and [13].

Finally, we discuss related work concerning attack grammars for fuzz testing of XSS attack vectors. In particular, such grammars were employed in [14] and [15] where an evolutionary and a learning approach to detect vulnerabilities has been employed, respectively.

III. Combinatorial Security Testing

A. *Combinatorial Testing*

Testing a SUT requires the existence of test cases and in particular a method capable of generating such test cases. For developing our testing framework we can also use methods that arise from the field of combinatorial testing, which is an effective testing technique to reveal errors in a given SUT, based on input combinations coverage. Combinatorial testing of strength t (where $t \geq 2$) requires that each t-wise tuple of values of the different system parameters is covered by at least one test case.

To design a test case, the tester identifies possible output values from each of the actions of the SUT. It is important to find a test case that is not too large, but yet tests for most of the interactions among the possible outputs in the action of the SUT. Recently, some researchers [16], [17], [18] suggested that some faults or errors in SUTs are a combination of a few actions when compared to the total number of parameters of the system under investigation.

Combinatorial testing is motivated by the selection of a few test cases in such a manner that good coverage is still achievable. The combinatorial test design process can be briefly described as follows:

1) Model the input space. The model is expressed in terms of parameter and parameter values.
2) The model is input into a combinatorial design procedure to generate a combinatorial object that is simply an array of symbols.
3) Every row of the generated array is used to output a test case for a SUT.

One of the benefits of this approach is that steps 2. and 3. can be completely automated. In particular, we used the ACTS combinatorial test generation tool [4] and subsequently the attack pattern-based methodology given in Section IV for these steps. ACTS is developed jointly by the US National Institute Standards and Technology and the University of Texas at Arlington and currently has more than 1400 individual and corporate users.

B. *Combinatorial Grammar for XSS Attack Vectors*

In this section, we consider a general structure for XSS attack vectors (test inputs) where each one of them is comprised of 12 discrete parameters where 6 of them are single-valued or have to satisfy certain constraints. Our combinatorial grammar is presented below in BNF form so as to be able to generate possible parameter values through ACTS. We would like to note that although attack grammars for XSS attack vectors have been given in [14], [15], this is the first time that such a grammar is used in terms of combinatorial modeling and our approach for *combinatorial security testing* is novel in that sense.

```
FOBRACKET ::= <
TAG ::= img | frame | src | script | body | HEAD |
    BODY | iframe | IFRAME | SCRIPT
FCBRACKET ::= >
QUOTE1 ::= ‘‘ | ‘ | null
SPACE ::= \n | \t | \r | \r\n | \a | \b | \c | _ |
    null
EVENT ::= onclick | onmouseover | onerror | onfire |
     onbeforeload | onafterload | onafterlunch |
    onload | onchange | null
SPACE2 ::= \n | \t | \r | \r\n | \a | \b | \c | _ |
    null
QUOTE2 ::= ‘‘ | ‘ | null
PAYLOAD ::= alert(1) | alert(0) | alert(document.
    cookie) | alert(”hacked”) | alert(’hacked’) |
    src=”http://www.cnn.com”&gt;
LOBRACKET ::= </
CLOSINGTAG ::= img | frame | src | script | body |
    HEAD | BODY | iframe | IFRAME | SCRIPT
LCBRACKET ::= >
```

BNF Grammar for XSS Attack Vectors

Based on the previously presented attack grammar we consider an XSS attack vector to be of the following form:

$$\text{AV} := \langle \text{FOBRACKET, TAG, FCBRACKET, QUOTE}_1\text{, SPACE, EVENT, SPACE2, QUOTE2, PAYLOAD, LOBRACKET, CLOSINGTAG, LCBRACKET} \rangle$$

The given parameter values are just a fragment of possible options. If the designer would like to generate a vast number of inputs, it is sufficient to just add parameter values in the given BNF for XSS attack vectors. Moreover, our grammar also satisfies some constraints using the constraint tool from ACTS. For example, we match the TAG parameter with the CLOSINGTAG parameter to always be able to produce valid inputs.

For the second step of the combinatorial test design process we use the notion of mixed-level covering arrays (a specific class of combinatorial designs). For the sake of complicity we provide below the definition of mixed-level covering arrays taken from [19] since this is the underlying generated structure in the ACTS tool:

Definition 1: A mixed-level covering array which we will denote as $MCA(t, k, (g_1, \ldots, g_k))$ is an $k \times N$ array in which the entries of the i-th row arise from an alphabet of size g_i. Let $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ and consider the subarray of size $t \times N$ by selecting rows of the MCA. There are $\prod_{i=1}^{t} g_i$ possible t-tuples that could appear as columns, and an MCA requires that each appears at least once. The parameter t is also called the strength of the MCA.

We would like to remark that this technique of discretizing the parameter values is referred to as input parameter modeling for combinatorial testing [20] and essentially enables the designer to choose the possible parameter values for the SUT. Thus, it is natural to define our attack grammar as a combinatorial one when the first one is used for input parameter modeling. Essentially, this means that given the t-wise interaction of the covering array we generate with ACTS all possible t-tuples of parameter values for a number of t total parameters in the SUT. This is another explanation for the strength t of the covering array where for any selection of t-rows each t-tuple appears at least once.

For all cases we shall consider in this paper, the parameters of the MCA are derived from the BNF combinatorial grammar according to the following formulation. The number of rows of the MCA equals to the number of types in the presented combinatorial grammar while the size of alphabets g_i of the MCA equals to the number of derivation rules per type. For example, all XSS attack vectors for the *src* tag when we want to test for pairwise interactions ($t = 2$) can be found in Table I.

IV. Attack Pattern-Based Testing

This paper takes the technique from [9], which elaborates a model-based testing approach for testing of web applications. The difference to other model-based approaches is the fact, that we do not use a model of the SUT to derive test cases but take as a model the definition of attack patterns from [21], where such a construct is specified by a goal, preconditions, actions and postconditions. In our case, the attack pattern is represented in form of a UML state machine and depicts the execution paths for one or more types of attacks by starting from the beginning of the testing process and finally concluding at its end. Between these two points lie a number of states and transitions with guards and effects specified as parts of these elements with variables and methods attached to transition labels. The test case generation is not the task here.

The meaning of this approach is to use a pattern, which should be applicable for attacking and thus testing of web applications with only a minimum of required tester interaction at the beginning of that process. The model itself is created manually in the Eclipse based open source tool-kit YAKINDU Statechart Tools[1] according to HTTP and web application structure, which relies on HTML. Also, the framework Web-Scarab[2] can be used before creating the attack pattern in order to get an overview about the communication structure between client and server. Because we want to test for XSS vulnerability, we implemented a checking mechanism, which will be described later in more detail. The model itself also comprises variables and method calls, which are assigned to the transition labels of the state machine. The states represent certain situations in the testing process, which are entered if its preconditions are satisfied. During execution as transitions are activated, the according methods and value manipulation is carried out automatically. In this paper, we use the attack pattern-based approach for the execution of test cases.

In order to cover HTTP-based communication between tester and SUT, a HttpClient[3] was implemented in Java for creating and reading of HTTP messages. The client is used in combination with the parser jsoup[4] for parsing HTTP responses and extracting its HTML elements, which will be critical in the detection of XSS vulnerabilities. Another important feature of HttpClient is the possibility to bypass communication via browser so potentially browser intern defense mechanisms are evaded as well. Also, some application specific constraints like the length of the input field are also bypassed so much longer inputs can be injected.

In general, XSS in web applications is found upon unsanitized user inputs, i.e. if a submitted script is processed unfiltered by an application, then its potentially harmful content is sent to a client and executed, eventually causing damage to him. An indicator for such vulnerability is a returned HTML element that can be parsed from the server's response because otherwise the script is filtered so the client might read only some non-executable code. Because of this fact, the goal of our approach is to inject such a string, which will go not sanitized through the application and to detect if it was manipulated when parsing the response.

In order to realize this approach, methods are implemented in Java, which will be called from the model during execution so these methods may accept incoming variable values and return new ones to the model back while running in the background. It should be noted again that all of the described entities are integrated into the Eclipse development environment.

[1]http://statecharts.org/
[2]https://www.owasp.org/index.php/Webscarab
[3]http://hc.apache.org/httpcomponents-client-ga/
[4]http://jsoup.org/

TABLE I
A SAMPLE OF XSS ATTACK VECTORS FROM $MCA(2, 12, (1, 1, 1, 1, 3, 3, 6, 9, 9, 10, 10))$

FOBRACKET	TAG	FCBRACKET	QUOTE1	SPACE	EVENT	SPACE2	QUOTE2	PAYLOAD	LOBRACKET	CLOSINGTAG	LCBRACKET
<	src	>	’	\r\n	onclick	\b	’	alert(document.cookie)	</	src	>
<	src	>	null	\a	onmouseover	\c	’’	alert(\hacked\")"	</	src	>
<	src	>	’’	\b	onerror	_	null	alert(’hacked’)	</	src	>
<	src	>	null	\c	onfire	null	’’	src=\http://www.cnn.com\">"	</	src	>
<	src	>	’	_	onbeforeload	\n	’’	alert(1)	</	src	>
<	src	>	’	nil	onafterload	\t	’’	alert(0)	</	src	>
<	src	>	’’	\n	onafterlunch	\r\n	’’	alert(document.cookie)	</	src	>
<	src	>	’	\t	onload	\a	’	alert(\hacked\")"	</	src	>
<	src	>	’’	\r	onchange	\b	’’	alert(’hacked’)	</	src	>
<	src	>	’	\r\n	nil	\r	’	alert(1)	</	src	>

One idea behind the approach is to offer the possibility for the model to be applicable for testing of several web applications without the need to make any additional changes in the model or to add new variables. Although, there is the provided functionality to add additional states and methods by the tester and to attach them to the model but this is of no concern in this work.

In the next section we describe the approach as well as the connection to the previous combinatorial part of this paper on a concrete example.

V. CASE STUDY

We demonstrate the implementation of both approaches by testing the web application BodgeIt. A possible attack pattern for testing for reflected and stored XSS is depicted in Figure 1. The model here is kept short in order to give an understandable overview of the approach so basically it is applied only to test for XSS although another attacks can be tested as well with expanding the statechart. Left in the window we see the defined interfaces, variables and method calls. Interfaces have no critical function but to specify what variables belong to what role, in our case either the attacker or the SUT. As mentioned before, the methods are implemented manually as part of the project and they have the task to dynamically return new values to the model, thereby directing which path the execution will follow.

The execution itself starts in the initial state and because there are neither guards nor actions specified for the first transition, the state $Started$ is activated. Until now, no methods or variables are referenced. In our approach, all generated inputs from ACTS are stored in a text file line by line. In fact, these strings represent the actual test cases. Now a method, which runs behind the model, reads and saves those strings from the file into an ArrayList and returns the number of inputs. The next transition checks this value and if the data structure isn't empty, the model proceeds further to the next one. At this moment we know that we have inputs in our program.

The next transition asks for the URL of the SUT to be submitted before an action can occur and when the tester provides the address, the method $connect$ is called with the URL as an argument. This one tries to connect to the given address by sending a request and returns the status code of the response and if the status is OK, then the execution continues into the next state. It should be noted that the tester can submit the URL at the beginning of the execution, i.e. before a precondition requests this from him.

In order to create a HTTP message with the necessary parameters, we parse the response for the request method, input fields and their values, which is done before activating $Parsed$ by the corresponding method. Also, this method counts separately all XSS-critical HTML elements like <*script*>, <*src*>, <*img*>, <*iframe*> etc. from the web page and stores their numbers.

Now, the attacking looks like follows. The tester submits manually some starting injection string that can be specified before reaching this area, which is submitted to the input fields of the SUT. This is carried out by the method $attackXSS$, which afterwards parses the response in search for unfiltered HTML elements by counting again the number of occurrences of critical elements and if the new values are greater than the previously ones obtained during $parseInitial$, a positive result is returned in form of a Boolean variable. In that case, back in the model a success counter is increased by one so we know that a vulnerability is detected. In both positive or negative conditions, the test case ends and the execution returns back to a previously active state and takes the next input string from the ArrayList with ACTS results. An additionally counter for the next in line is updated in the model after every test case execution so the program always picks the next string. Then, this new input is submitted against the SUT in search for unfiltered input. This process will continue as long as inputs are available but then, the execution of the model finishes. In the end, the counter value indicates how many inputs were successfully categorized as security leaks.

Generally, this mechanism enables the detection of both reflected and stored XSS by assuming that an additionally found HTML element is in fact an indicator for an injection possibility.

VI. EVALUATION

For the evaluation three different web applications were deployed locally and tested against both types of cross-site scripting, with every one of them being checked manually before the actual testing. For combinatorial interaction strength 2, ACTS threw out 114 inputs and respectively 1031 and 8332 strings for strength 3 and strength 4. The obtained results of the evaluation are depicted in Table II. Also, it should be mentioned that we tested input fields with textual or password values as well as textarea tags.

Also, DVWA and Mutillidae include more security levels, which can be selected by the user. Testing on each of these levels will differ with regard to built-in filtering mechanisms,

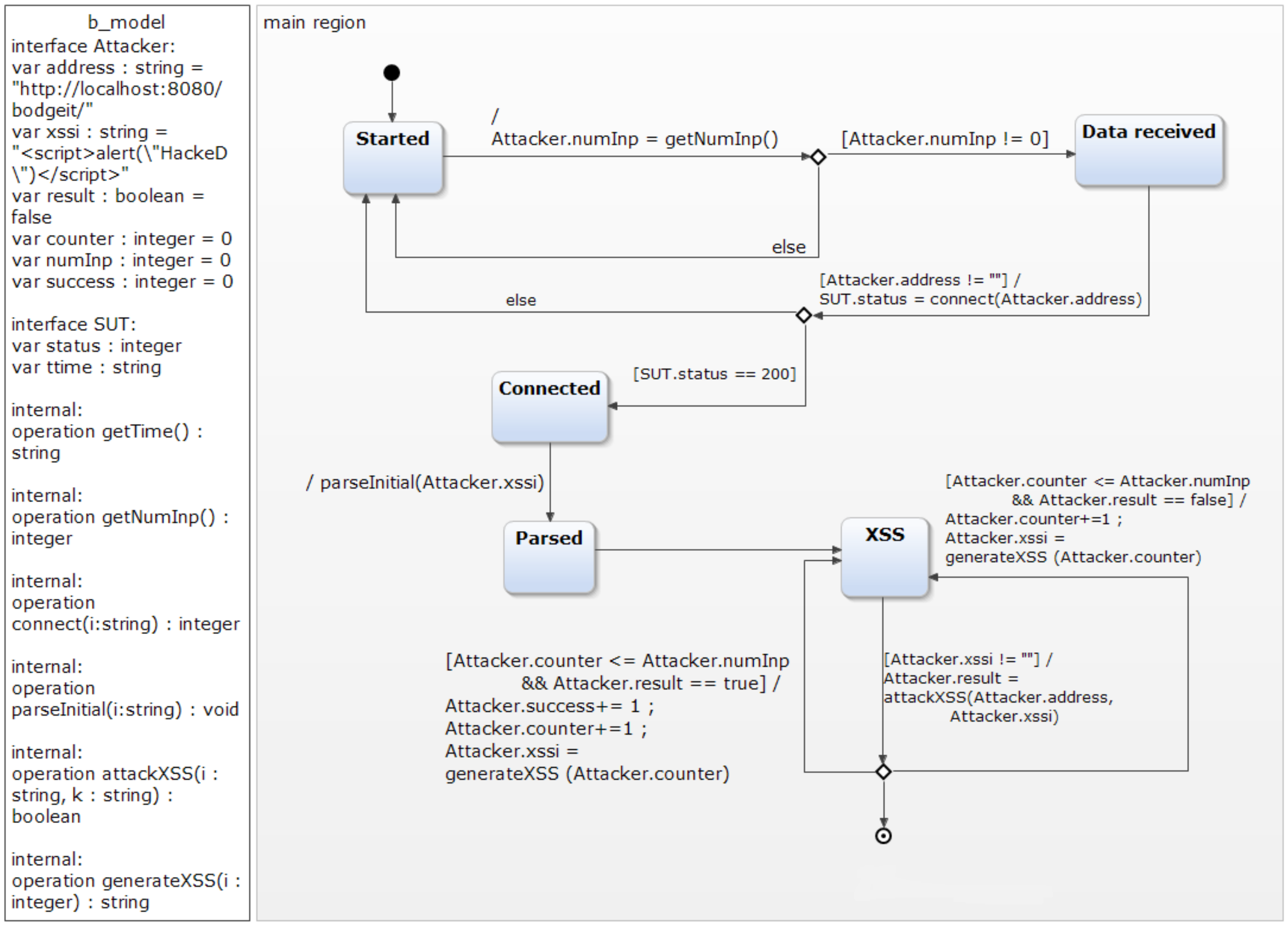
b_model
interface Attacker:
var address : string = "http://localhost:8080/bodgeit/"
var xssi : string = "<script>alert(\"HackeD\")</script>"
var result : boolean = false
var counter : integer = 0
var numInp : integer = 0
var success : integer = 0
interface SUT:
var status : integer
var ttime : string
internal: operation getTime() : string
internal: operation getNumInp() : integer
internal: operation connect(i:string) : integer
internal: operation parseInitial(i:string) : void
internal: operation attackXSS(i : string, k : string) : boolean
internal: operation generateXSS(i : integer) : string
main region
Started
/ Attacker.numInp = getNumInp()
[Attacker.numInp != 0]
Data received
else
[Attacker.address != ""] / SUT.status = connect(Attacker.address)
else
Connected
[SUT.status == 200]
/ parseInitial(Attacker.xssi)
Parsed
XSS
[Attacker.counter <= Attacker.numInp && Attacker.result == false] / Attacker.counter+=1 ; Attacker.xssi = generateXSS (Attacker.counter)
[Attacker.counter <= Attacker.numInp && Attacker.result == true] / Attacker.success+= 1 ; Attacker.counter+=1 ; Attacker.xssi = generateXSS (Attacker.counter)
[Attacker.xssi != ""] / Attacker.result = attackXSS(Attacker.address, Attacker.xssi)

Fig. 1. Attack pattern for BodgeIt

which become even more restrictive with each higher level so these applications had to be tested against all of them. BodgeIt, on the other side, doesn't have any additional security features.

In this paper we assume that the most important indicator for XSS vulnerability is the detection of an unfiltered HTML element by the parser, e.g. *<script>*, *<img>* etc., which was sent from the tester to the application as the input. In contrast to manual testing, we encountered several issues during the process about the structure and meaning of successful vulnerability detection. Before doing the automated tests, many of the generated inputs from ACTS were used by the testers in order to see how the SUT reacts by executing them in the browser. During that process we also drew the conclusion, that Google Chrome is more resistant to malicious inputs than Mozilla Firefox regarding the filtering of inputs on browser side so many scripts which were executed in Firefox, could not be triggered in Chrome. For this sake, another interesting fact is that our program reported more encounters of critical HTML elements than when executed by a browser.

In the table we can observe that in DVWA not a single breakthrough was reported on the medium and hard levels. On the lowest security level the got the same results for both persistent and stored XSS, so we assume that the software uses identical defense mechanisms for these cases although this seems not to be the case with the higher level, where type-2 XSS could not be triggered at all.

On the hand, Mutillidae and BodgeIt have been hacked on both levels for both types of XSS.

Our observations led us to the conclusion that only certain tags could go unfiltered, especially $script$, src and $iframe$ while others like $body$, $head$ and $frame$ were not able to cause a security break. This leads to the obvious conclusion that such applications need more protection against these critical tags and keywords.

Also, our specified attack grammar in ACTS is so far successful for script tags in case no filtering is applied, so it actually may be possible to put a more malicious code inside a script by using the same structure but by expanding the grammatical content between the tags. On the other hand, we probably need a redefinition of unsuccessful elements. All this does not downgrade the using of combinatorial testing but eventually reveals something about the mechanism of designing grammars and payloads for XSS, which might be expanded further.

The reason for the overall results may be the fact that the structure of the generated inputs by ACTS is relatively

TABLE II
EVALUATION RESULTS

Strength	Application	Type of vulnerability	Difficulty Level	Execution time (s)	# of successful injections	% coverage
2	BodgeIt	RXSS	-	30.00	69	60.53
		SXSS	-	31.80	56	49.12
3		RXSS	-	244.60	619	60.04
		SXSS	-	242.80	512	49.66
4		RXSS	-	1788.70	4991	59.90
		SXSS	-	1950.80	4157	49.89
2	DVWA	RXSS	1	27.40	69	60.53
		SXSS	1	28.50	69	60.53
		RXSS	2	30.70	56	49.12
		SXSS	2	30.60	0	0.00
3		RXSS	1	281.80	619	60.04
		SXSS	1	284.90	619	60.04
		RXSS	2	269.30	512	49.66
		SXSS	2	289.20	0	0.00
4		RXSS	1	2284.30	4991	59.90
		SXSS	1	3200.60	4991	59.90
		RXSS	2	2684.19	4157.00	49.89
		SXSS	2	3010.10	0	0.00
2	Mutillidae	RXSS	1	56.60	69	60.53
		SXSS	1	61.10	41	35.96
		RXSS	2	74.20	69	60.53
		SXSS	2	80.80	41	35.96
3		RXSS	1	514.00	619	60.04
		SXSS	1	536.30	302	29.29
		RXSS	2	625.90	619	60.04
		SXSS	2	700.30	302	29.29
4		RXSS	1	3990.90	4991	59.90
		SXSS	1	4451.70	2398	28.78
		RXSS	2	5246.00	4991	59.90
		SXSS	2	5586.50	2405	28.86

manageable according to the specified parameter segments for the generation process. For example, we specified that all of the inputs begin with the opening tag and conclude with its closing counterpart and also use the usual HTML element names. We also implemented the detection mechanisms in our program in order to be effective against the same symbols and HTML elements. In all the cases where the SUT didn't have any intern filtering mechanisms specified against these symbols, the submitted element is returned and parsed successfully without being filtered by the application. However, if the system already has some prevention mechanisms like the evasion of specific tags or keywords from the input, then all incoming injections with these symbols were rejected. For that reason, no testing quality could be achieved just by increasing the number of inputs as long as some different combinatorial syntax isn't defined. In fact, we got high coverage rates in all cases where this filtering hasn't been applied.

Another very interesting fact is that for every combinatorial interaction strength we got the same coverage of positive results, i.e. vulnerability detections, no matter how many inputs we submitted. Also, we tested BodgeIt and Mutillidae with inputs from strength 5 with 48755 inputs but then only to reaffirm the same conclusion.

From a purely combinatorial point of view we draw some additional conclusions for the same coverage of positive results. Firstly, we deduce that the interaction of the involved parameters is independent of the SUTs we evaluated. In other words, if we take a close look on the evaluation results we see that an interaction of two modeled parameters ($t = 2$) is sufficient for a tester to penetrate the specific web applications (at least on the first security levels where applicable). As already discussed in Section III, increasing the strenght of the covering arrays in ACTS for $t = 3$ and $t = 4$ implies an increase on the number of interactions of the parameters of the generated inputs. Having this in mind, obtaining the same coverage percentage leads us to the conclusion that every possible positive input has been generated already for strength $t = 2$ (and reproduced for higher strengths).

As usually is the case when considering the mathematical modeling of complex systems (and such are the SUTs we evaluated) there are some certain advantages and disadvantages of our approach for attack pattern-based combinatorial testing. One positive aspect of the presented methodology is that the (general) combinatorial testing oracle from [22] can also be used for security testing of web applications. Recall that this oracle is based on first testing all two pairwise interactions ($t = 2$). Then the tester continues by increasing the interaction strength t until no further errors (injections in our case) are detected by the t-way tests (inputs), and finally optionally tries $t + 1$ and ensures that no additional errors are detected. The main motivation for combinatorial testing is the reduction of the search space while still being able to test for t-way interactions. The latest is also a sufficient condition for the successful application of combinatorial testing. We consider to achieve this milestone as for all tested SUTs, a significant reduction on the number of possible inputs was accomplished. For example, with our grammar we test all 2-way interactions with 114 inputs out of all 158799 possible inputs when considering exhaustive search of its parameters. This implies a reduction of 99.93% of the total search space produced by the combinatorial grammar while still being able to penetrate the aforementioned SUT.

This efficiency on test execution however does not come with some drawbacks. We mentioned that a more complex combinatorial grammar is needed to break through the higher security levels of the SUTs. Unfortunately, adding more values in the parameters of the grammar would contribute to an exponential growth of the computational time required to generated the required covering arrays by ACTS. One possible way to overcome this shortcoming would be to consider adding more parameters in the grammar instead of parameter values as this would increase the complexity only by a logarithmic scale. We conclude here the evaluation on the combinatorial part of our methodology and consider the last remark as a starting point for further research on the cross-fertilization of the fields of combinatorial testing and web application security.

VII. Conclusion and Future Work

In this work we introduced a combined approach, which comprises the area of combinatorial testing with the emphasis on test case generation for XSS attacks and the attack pattern-based testing technique for test case execution against web applications. The generated strings were automatically tested against three web applications with different combinatorial interaction strength with highly promising initial results.

The main task for expanding this technique into a more general approach for automatically XSS detection is to put more emphasis on the extension of an attack grammar for XSS although this presents a greater challenge because of the variety of possible inputs and an undefined structure of such strings.

Also, quite different results might be generated by setting more constraints on the parameter structure in ACTS. In addition, mutation functions may be applied from the attack pattern model on the generated inputs, e.g. by applying different encoding types or evasion of special symbols so that as many as possible vulnerability sources may be detected in some software.

Acknowledgement

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 ITEA-2 (DIAMONDS) and the Austrian COMET Program (FFG). In addition, the work of the second author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (FP7/2007-2013) under *grant agreement* no. 246016.

References

[1] "Burp suite," http://portswigger.net/burp/, accessed: 2014-01-28.
[2] "Ibm security appscan," http://www-03.ibm.com/software/products/en/appscan, accessed: 2014-01-28.
[3] "sqlmap," http://sqlmap.org/, accessed: 2014-01-28.
[4] NIST, *User Guide for ACTS*. [Online]. Available: http://csrc.nist.gov/groups/SNS/acts/documents/acts_user_guide_v2_r1.1.pdf
[5] M. Brcic and D. Kalpic, "Combinatorial testing in software projects," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 1508–1513.
[6] 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.
[7] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.
[8] C. Phillips and L. Swiler, "A graph-based system for network vulnerability analysis," in *ACM New Security Paradigms Workshop*, 1998, pp. 71–79.
[9] 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.
[10] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.
[11] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," in *Proceedings of the Model-Based Testing Workshop at ETAPS 2012. EPTCS*, 2012, pp. 1–12.
[12] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, March 2011, pp. 531–533.
[13] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, USA, 2008.
[14] 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.
[15] 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.
[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, 1997.
[17] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.
[18] 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*, 2013, pp. 370–375.
[19] C. J. Colbourn, "Covering arrays," in *Handbook of Combinatorial Designs*, 2nd ed., ser. Discrete Mathematics and Its Applications, C. J. Colbourn and J. H. Dinitz, Eds. Boca Raton, Fla.: CRC Press, 2006, pp. 361–365.
[20] 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.
[21] 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.
[22] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.