# Evaluation of the IPO-Family Algorithms for Test Case Generation in Web Security Testing

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

Bernhard Garn
SBA Research
A-1040 Vienna, Austria
bgarn@sba-research.org

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*—**Security testing of web applications remains a major problem of software engineering. In order to reveal vulnerabilities, testing approaches use different strategies for detection of certain kinds of inputs that might lead to a security breach. Such approaches depend on the corresponding test case generation technique that are executed against the system under test. In this work we examine how two of the most popular algorithms for combinatorial test case generation, namely the IPOG and IPOG-F algorithms, perform in web security testing. For generating comprehensive and sophisticated testing inputs we have used input parameter modelling which includes also constraints between the different parameter values. To handle the test execution, we make use of a recently introduced methodology which is based on model-based testing. Our evaluation indicates that both algorithms generate test inputs that succeed in revealing security leaks in web applications with IPOG-F giving overall slightly better results w.r.t. the test quality of the generated inputs. In addition, using constraints during the modelling of the attack grammars results in an increase on the number of test inputs that cause security breaches. Last but not least, a detailed analysis of our evaluation results confirms that combinatorial testing is an efficient test case generation method for web security testing as the security leaks are mainly due to the interaction of a few parameters. This statement is further supported by some combinatorial coverage measurement experiments on the successful test inputs.**

***Keywords—Combinatorial testing, constraints, IPO-Family algorithms, 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. Naturally, injection attacks are remote exploits 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 [1]. In this paper, 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). These two types are also referred to as type-1 and type-2 XSS, respectively.

Security testing is meant to support vulnerability detection, and for this task several approaches and tools have been developed in the past. We depict the most important of them in the related work section. Manual testing tools provide mechanisms for a user to adapt the testing framework to the *System Under Test* (SUT) by choosing elements to be tested. An automation of this process is certainly desirable. However, complete automation as of today is still under constant development, even though there are some partly automated tools [2], [3] that support professional penetration testers. It is important to note that any testing methodology relies on effective and high-quality test case generation, regardless of the subsequent manual or automatic execution.

In [4] a novel method was described that combines an adapted combinatorial testing approach for the generation of test inputs according to a specified XSS grammar with a technique, which uses patterns of attacks for the execution of test cases called attack pattern-based testing. Because the initial results for tested applications were quite optimistic, we improved that approach by extending the input grammar and adding constraints between the input parameter values in [5].

We rely on the combinatorial test generation tool ACTS [6] for input model specification (in terms of input parameter modelling [7]) and generation of a combinatorial object. We reuse the modelling of XSS attack vectors given in [5], where the main goal was to compare combinatorial testing versus fuzzers. Then, we derive four concrete sets of test suites as inputs for SUTs from the generated combinatorial structures, each two of them generated via the IPOG and the IPOG-F algorithm, respectively, also depending on whether they have constraints or not. The subsequent test execution is performed by the automated method first given in [4].

In this paper, we change the scope of our research and focus on comparing, for the first time, the efficiency of the IPO-family algorithms when the generated test inputs are submitted against (the same) web applications in order to reveal security vulnerabilities. Furthermore, the paper includes a detailed case study for testing various SUTs against our automated testing method and a detailed evaluation and comparison of the two sets of test suites is presented. We also draw useful conclusions that indicate that combinatorial testing is an efficient approach for security testing, building on the comparison of the test results of the testing methods given in [4], [8], [5]. Most important we present an extensive analysis in terms of (total) vulnerabilities found by our generated test suites using the

The work of the third author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

Authors are listed in alphabetical order.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
4th International Workshop on Combinatorial Testing (IWCT 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

combinatorial coverage measurement (CCM) tool [9], where it is revealed that the indicated security leaks are mainly caused by the interaction of few parameters.

The main *contributions* of this paper are summarized as follows:

- A detailed comparison of the IPOG and IPOG-F test case generation algorithms for their application in web security testing. The test inputs (XSS attack vectors) are generated in an automated way;
- Detailed evaluation of a case study for web security testing via an automated test execution technique using also combinatorial coverage measurement methods.

The paper is structured as follows: Section I provides related research from both combinatorial testing as well as model-based techniques for testing. In Section II, we review combinatorial testing and its potential contribution to security testing. Afterwards, Sections IV and III revisit the attack pattern-based approach for test case execution and provide a working example in a case study, respectively, for web security testing. Then Section V discusses the testing results of our automated approach against several web applications in conjunction with combinatorial coverage measurement experiments and finally, Section VI concludes the work.

## I. Related Work

For important works in combinatorial testing that relate to our research we refer to Section II and cited references therein, while for a general treatment of the field of combinatorial testing we refer the interested reader to the recent surveys of [10] and [11].

However, we give here a flavor of the many different application areas involved, in order to exhibit that while combinatorial testing is a specialized methodology for test case generation, its application to the domain of software testing is of current and growing interest.

For example, three different case studies of combinatorial testing methods in software testing have been given in [12]. Applying combinatorial testing to the Siemens Suite and testing ACTS with ACTS have been presented in [13] and [14], respectively. A case study for pairwise testing through 6-way interactions of the Traffic Collision Avoidance System (TCAS) has been presented in [15], while in [16] a study was conducted to replicate the Lockheed Martin F-16 combinatorial test study in a simplified manner. Moreover, a proof-of-concept experiment using a partial $t$-wise coverage framework to analyze integration and test data from three different NASA spacecrafts has been presented in [17]. Finally, combinatorial testing on ID3v2 tags of MP3 files was given in [18] while a case study for evaluating the applicability of combinatorial testing in an industrial environment was presented in [19].

Some of the following works describe ideas that use models of attacks in order to breach software security.

The technique for test case execution in this paper is built upon former works of the authors from [20] and [21]. They describe a method called attack pattern-based testing, which is meant to execute test cases automatically against a SUT. The process is guided by a model of the attack, which is implemented as a UML state machine. Preconditions and actions are implemented as parts of the statechart whereas methods return new values during execution on a lower level. While traversing the model, new variable values might change the path of execution according to satisfied preconditions. This approach also encompasses detection mechanisms for both SQL injections and reflected as well as stored XSS. It offers the advantage of being applicable to any web application while demanding only negligible user interaction.

However, the idea of depicting attacks in a model was suggested by Phillips and Swiler in [22]. There they introduced a way to generate a graphical representation of an attack. For this case, an attacker profile, configuration file and attack templates have to be generated. The approach proceeds backwards, starting from the goal node and branching paths where the templates satisfy specific preconditions. The most obvious difference of our model-based approach lies in the way the model is generated. Also, the capability for XSS detection distinguishes our method from other works.

The authors from [23] describe a method that relies on attack models in form of a UML statechart. They define a testing tool that depends upon libraries of attack vectors in order to execute test cases. However, the main difference to our approach is the manner in which tests are generated but also the technical implementation of the method. In addition, in our work we consider execution and detection mechanisms for XSS.

A comparison of several penetration testing tools is given in [24]. The authors compare commercial as well as open source penetration testing tools by testing several web applications. One of the SUT was Mutillidae, which is also tested in this paper as part of the evaluation. However, it should be mentioned that our approach is able to report more cases in which XSS could be detected.

A more detailed picture about XSS is given by Fogie et al. in [25] whereas model-based testing is explained in [26].

Attack grammars for XSS have been also used for fuzzing in [3], [27] and [28]. The authors of the last two works apply evolutionary approaches and learning in order to detect potential vulnerabilities. However, in these works the attack grammars had to be manually designed. The automation of this process still remains an active research problem. In contrast to these works, the test execution in this paper is guided solely by the model of an attack.

Finally, combinatorial testing has recently been employed as a method to model XSS attack vectors in [4], [8]. In detail, in [4] a novel combinatorial testing technique for generation of XSS attack vectors was first defined while in [8] the applicability of the previous technique has been further demonstrated by relaxing constraints and modelling white spaces in the attack grammar. Moreover, in [5] fuzz testing approaches for generating XSS attack vectors where compared to combinatorial testing techniques and the evaluation results showed that combinatorial testing is a viable alternative to fuzzers for revealing XSS vulnerabilities.

We would like to note that, in all of the previous works the focus was to compare manual to automated penetration testing tools and the underlying combinatorial test generation algorithm was IPOG. The main difference to this work is the inclusion of the IPOG-F algorithm for generating the test suites and a comparison between these two algorithms of the IPO-family applicable in web security testing, for the first time.

## II. Combinatorial Security Testing

### A. Combinatorial Testing

Combinatorial testing is motivated by the selection of a few test inputs in such a manner that good coverage in regard to the total modelled discretized input space is still achievable. It is also important to mention that the notion of coverage varies with respect to different metrics in security testing (c.f. Section II-D, for more details). Recently, some researchers [29], [30], [31] suggested that some faults or errors in SUTs are due to a combination of few parameters when compared to the total number of components of the SUT.

A more thorough description of the combinatorial test design process can be found in [32] and in [4], [8] for a more related usage to security testing.

In particular, we used the ACTS combinatorial test generation tool [6] for automated test generation of inputs and subsequently the attack pattern-based methodology given in Section IV for test execution. 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[1].

We have used the IPOG test generation algorithm [33] and its refinement IPOG-F [34] to generate the test suites, as these are implemented in ACTS. Both algorithms are a generalization of the in-parameter-order (IPO) strategy first introduced by Lei and Tai [35]. A detailed description of the IPO-family can be further found in [32] and [36].

### B. Combinatorial Grammar for XSS Attack Vectors: A Review

In this section, we review the general structure for XSS attack vectors (test inputs) where each one of them is comprised of 11 discrete parameters (types) and discussed in detail below. This structure builds upon a combinatorial grammar given in [4] by modelling whitespaces and executable JavaScript that can appear in an XSS attack vector but also extends the ones given in [8], [5]. Note that [5] also includes 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 [37], [3].

For the sake of completeness we present below a fragment of our combinatorial grammar, denoted by G, 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)::=␣<scr<script>ipt>␣|␣<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 [37]. 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 previously presented attack grammar and lessons learned from past experiences we designed the following form of an XSS attack vector (AV):

$$\textbf{(JSO,WS1,INT,WS2,EVH,WS3,PAY,WS4,PAS,WS5,JSE)}. \tag{1}$$

The description of the types in the previous AV can be found in [8], [5] and we do not include them here as this is not the focus of the current paper. However, we would like to mention that the AV was designed in such a way in order to produce valid JavaScript code when this is injected into SUT parameters.

For the next step of the combinatorial test design process we use the notion of mixed-level covering arrays (a specific class of combinatorial designs). We provide below the definition of mixed-level covering arrays taken from [32] since this is the underlying generated structure in the ACTS tool:

*Definition 1:* A mixed-level covering array which we will denote as $\mathrm{MCA}(t, k, (g_1, \ldots, g_k))$ 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 modelling for combinatorial testing [32], [7] 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 modelling. 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

[1] http://csrc.nist.gov/groups/SNS/acts

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 types that form an XSS attack vector according to the following formulation. The number of rows $k$ of the MCA equals the number of types in the presented form of the attack vector while the size of the alphabets $g_i$ of the MCA equals the number of different values per type.

### *C. Adding Constraints*

In this section we mention that constraints were added to our XSS attack grammar. The motivation for this reason rises from the fact that when testing real-world systems adding constraints in the respective model may produce test suites with better quality and also considerably reduce the search space. This approach for combinatorial testing has been followed for example in [38], [14], [39].

We would like to note that although attack grammars for XSS attack vectors have been given in [4], [8], [5], [3], [27], [28], this is the first time that constraints are imposed on such attack grammars in terms of combinatorial modelling when the model is input to the IPOG-F algorithm.

The full set of 28 constraints for our grammar and its rationale in web security testing has been given in [5] and we avoid repeating it here. However, we give an example of some of them using the constraint support from ACTS where the symbol => denotes an implication and the symbol || an OR statement.

```
(JSO=5) => (JSE=5 || JSE=6 || JSE=7 || JSE=8 || JSE=9)
(EVH=1) => (PAY=12 || PAY=14 || PAY=17 || PAY=18 || PAY=19)
(WS1=WS2 && WS2=WS3 && WS3=WS4 && WS4=WS5)
```

We will denote this grammar with `G_c` to distinguish it from `G` when constraints are enforced.

### *D. 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 were also adopted by security researchers [40] and some new ones have been proposed [41].

In the context of automated security testing for web applications (c.f. [41]), we defined in [5] 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:

$$\text{ER} = \frac{\#\text{ Attack vectors that exploit an XSS vulnerability}}{\text{Total number of attack vectors per test suite and SUT}} \tag{2}$$

In this work, we have also used combinatorial coverage measurement metrics (which differ to the ER) to determine the quality of the successful attack vectors. In particular, we are interested on the number of $t$-way combinations fully covered in passing tests. This notion of simple $t$-way combination coverage [32], which can be computed by the CCM tool [9], is defined as the proportion of fully covered $t$-way combinations of given $k$ parameters in a test suite. By full covered $t$-way combinations we mean that all variable-value configurations for a set of $t$ parameters appear (at least once) in the test suite.

## III. 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].

Webgoat, Mutillidae and DVWA, were already tested in [4], [8] while WebGoat[4], Gruyere[5] recently in [5]. 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. However, in all of the previous case studies the test suites have been generated with the IPOG algorithm. In this case study, we have regenerated the test suites for IPOG and also created new ones with the IPOG-F algorithm. Moreover, in contrast to [8] we avoided using different test execution methods to keep the complexity of the problem to a reasonable level.

This was needed as one of the goals of our case study is to investigate which of the two algorithms of the IPO family, namely IPOG and IPOG-F, generates better quality test suites w.r.t. to XSS vulnerability detection. In particular, we want to compare the exploitation rate of the test suites focused on triggering XSS exploits (higher exploitation rate is better). Secondly, we are also interested to investigate the simple $t$-way combination coverage of passing tests to determine whether the revealed security leaks that are caused by the interaction of few parameters are invariant or not to the different parameter values per parameter. To the best of our knowledge, this is the first time that such kind of combinatorial coverage measurement metrics are applied to web security testing.

The SUTs were tested with different sets of test suites, produced by the ACTS tool and based on the combinatorial grammar given in Subsection II-B. The difference between the four sets of test suites rely on imposing various constraints on the different types of the attack grammar and the underlying combinatorial test case generation algorithm used. We also want to investigate whether in our experiments we can confirm the findings of [38], which state that imposing constraints on (models of) real-world applications makes higher strength combinatorial interaction testing feasible.

## IV. Test Execution Method

In this section, we provide some details about the test execution method used in our case study regarding its procedure, functionality and test oracle applicable when testing for XSS vulnerabilities.

[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/

**Attack Pattern-Based Combinatorial Testing.** This testing method relies on attack patterns, i.e. graphical representations of an attack. These guide the testing process during the execution of the program. The idea to apply patterns as a description of an attack has been introduced by Moore et al. [42]. In our case, the attack is depicted as an UML state machine with corresponding states and transitions. However, the model itself is created manually accordingly to the authors knowledge about the nature of the attacks. In fact, this graphical depiction represents an abstract test case, whereas the concrete values are implemented on a lower level. The execution starts in an initial state and eventually terminates in a final one.

Technical details about the implementation and technical realization of the approach are demonstrated in further detail in previous works of the authors [20], [21], [4].

**Test oracle.** In order to give a verdict whether an attack vector was able to exploit a vulnerability, corresponding detection mechanisms have been implemented. The system parses the response of a SUT after submitting a malicious input and searches for unintended outputs. In our case, an additional HTML element in the HTTP response serves as an indicator for a security leak. Usual values for these elements are <*script*>, <*img*> etc. If such code has been encountered, a vulnerability detection is reported. It should be mentioned that this mechanism can be applied for both types of XSS.

**Automated testing procedure.** In this paper, web applications are tested against reflected as well as stored XSS. Both attacks are specified in one attack model. Since the concrete implementation with corresponding variables and method calls are already set up, the tester is asked only for a minimum of interaction in order to start the execution. For demonstration purpose, we explain the approach while testing against one of the SUTs, namely `Gruyere`.

The program execution starts in the initial state of the model and proceeds further by trying to establish a connection with the SUT. Later during the process, the occurrence of various HTML elements in the system under test is counted. Then, the first test input from the list is submitted as part of a HTTP request against a specific part of the SUT. Afterwards, the incoming HTTP response is parsed by the program, thus recalculating the number of HTLM elements. If a discrepancy between the original number of occurrences and the new values of the individual elements occurred, we report the detection of a vulnerable element in the website. The corresponding test input and its critical elements are saved so that the tester can get an impression about what type of security leak should be expected. In order to give an example, we submit the following vector in order to test against stored XSS:

```
<script '> onLoad( alert(XSS) '> '\>
```

In case that a website does not have implemented filtering mechanisms for user inputs, the script will be processed unfiltered. Then, the original code will be sent back in its original form and executed on the client's side in which case a positive test is reported. However, in this case the SUT omits the critical <*script*> element, so it returns:

```
onLoad( alert(XSS) '> '\>
```

In this case, the response returns with a manipulated version of the submitted input with no valid HTML code to parse. An execution of the code is prevented and a failure is reported. Generally speaking, the success of an input depends on its own structure but also on the nature of the attacked SUT. After the test has been finished, the execution returns to a previously active state in the model and takes a new input from the list. The attack process is continued as long as additional entries are available in the list of input strings. Otherwise the execution terminates in the final state.

## V. Evaluation

As described in Section II-B we used an XSS grammar for the input model in ACTS and generated inputs for combinatorial interaction strengths $t \in \{2,3,4\}$ twice for each algorithm, where the first dataset consists of attack vectors that were created without setting any constraints on the data structure, while the second input file comprises attacks which were generated according to constraints mentioned in Section II-C. We tested all mentioned applications against all attack inputs and evaluated the respective results.

We note that the cardinality of the total space is $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 9 \times 11 \times 14 \times 15 \times 23 = 348585930$ tests. When generating the test suites using our model as input to a $MCA(t, 11, (3,3,3,3,3,3,9,11,14,15,23))$, for $t \in \{2,3,4\}$, we see a reduction of $\approx 99.99\%$ of the total search space. In addition, from Table I. it is evident that the test suites generated by the IPOG-F are smaller than the ones produced by the IPOG algorithm, for our model without constraints and the situation is reversed when these are enforced. Regarding the usage of constraints, enforcing them in test case generation makes the test suites even smaller in both cases. Finally, we note that the test generation in ACTS for all test suites was quite fast in a normal workstation, ranging from some seconds to a couple of minutes for increasing strength.

All test suites were executed using the method from Section IV. In order to draw a meaningful comparison, we tested the same parts of a SUT with all test suites, which were input fields with textual and password values or textarea tags but we also attached attack vectors to the URL paths without any variable binding.

### *A. Exploitation Rate of SUTs and their Relation to the IPO-Family*

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 for the IPOG and IPOG-F algorithms. The evaluation results are depicted in Table I for the XSS combinatorial grammar we have used (with and without constraints on the parameter values) when the generated vectors are tested against the SUTs described in the case study.

In particular, in Table I we give information about the combinatorial interaction strength (Str.), the SUT (App), the input field 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

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

| SUT parameters | | | | Str. | G | | | | G_c | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | IPOG | | IPOG-F | | IPOG | | IPOG-F | |
| App | DL | VT | inp_id | | ER | % ER | ER | % ER | ER | % ER | ER | % ER |
| Mutillidae | 0 | RXSS | 1 | 2 | 111/345 | 32.17 | 102/345 | 29.57 | 116/250 | 46.40 | 121/252 | 48.02 |
| Mutillidae | 0 | RXSS | 1 | 3 | 1580/4875 | 32.41 | 1561/4830 | 32.32 | 836/1794 | 46.59 | 950/2012 | 47.22 |
| Mutillidae | 0 | RXSS | 1 | 4 | 17344/53706 | 32.29 | 17127/53130 | 32.24 | 3974/8761 | 45.36 | 4449/9760 | 45.58 |
| Mutillidae | 0 | RXSS | 2 | 2 | 129/345 | 37.39 | 136/345 | 39.42 | 116/250 | 46.40 | 123/252 | 48.81 |
| Mutillidae | 0 | RXSS | 2 | 3 | 1849/4875 | 37.93 | 1822/4830 | 37.72 | 839/1794 | 46.77 | 942/2012 | 46.82 |
| Mutillidae | 0 | RXSS | 2 | 4 | 20135/53706 | 37.49 | 19957/53130 | 37.56 | 4108/8761 | 46.89 | 4667/9760 | 47.82 |
| Mutillidae | 0 | RXSS | 3 | 2 | 0/345 | 0.00 | 0/345 | 0.00 | 0/250 | 0.00 | 0/252 | 0.00 |
| Mutillidae | 0 | RXSS | 3 | 3 | 0/4875 | 0.00 | 0/4830 | 0.00 | 0/1794 | 0.00 | 0/2012 | 0.00 |
| Mutillidae | 0 | RXSS | 3 | 4 | 0/53706 | 0.00 | 0/53130 | 0.00 | 0/8761 | 0.00 | 0/9760 | 0.00 |
| Mutillidae | 1 | RXSS | 1 | 2 | 111/345 | 32.17 | 102/345 | 29.57 | 116/250 | 46.40 | 121/252 | 48.02 |
| Mutillidae | 1 | RXSS | 1 | 3 | 1580/4875 | 32.41 | 1561/4830 | 32.32 | 836/1794 | 46.59 | 950/2012 | 47.22 |
| Mutillidae | 1 | RXSS | 1 | 4 | 17344/53706 | 32.29 | 17127/53130 | 32.24 | 3974/8761 | 45.36 | 4449/9760 | 45.58 |
| Mutillidae | 1 | RXSS | 2 | 2 | 129/345 | 37.39 | 136/345 | 39.42 | 116/250 | 46.40 | 123/252 | 48.81 |
| Mutillidae | 1 | RXSS | 2 | 3 | 1849/4875 | 37.93 | 1822/4830 | 37.72 | 839/1794 | 46.77 | 942/2012 | 46.82 |
| Mutillidae | 1 | RXSS | 2 | 4 | 20135/53706 | 37.49 | 19957/53130 | 37.56 | 4108/8761 | 46.89 | 4667/9760 | 47.82 |
| Mutillidae | 1 | RXSS | 3 | 2 | 0/345 | 0.00 | 0/345 | 0.00 | 0/250 | 0.00 | 0/252 | 0.00 |
| Mutillidae | 1 | RXSS | 3 | 3 | 0/4875 | 0.00 | 0/4830 | 0.00 | 0/1794 | 0.00 | 0/2012 | 0.00 |
| Mutillidae | 1 | RXSS | 3 | 4 | 0/53706 | 0.00 | 0/53130 | 0.00 | 0/8761 | 0.00 | 0/9760 | 0.00 |
| BodgeIt | 0 | RXSS | 1 | 2 | 198/345 | 57.39 | 201/345 | 58.26 | 145/250 | 58.00 | 153/252 | 60.71 |
| BodgeIt | 0 | RXSS | 1 | 3 | 2842/4875 | 58.30 | 2842/4830 | 58.84 | 1073/1794 | 59.81 | 1207/2012 | 59.99 |
| BodgeIt | 0 | RXSS | 1 | 4 | 31441/53706 | 58.54 | 31120/53130 | 58.57 | 5366/8761 | 61.25 | 6084/9760 | 62.34 |
| BodgeIt | 0 | RXSS | 2 | 2 | 131/345 | 37.97 | 135/345 | 39.13 | 97/250 | 38.80 | 106/252 | 42.06 |
| BodgeIt | 0 | RXSS | 2 | 3 | 1890/4875 | 38.77 | 1888/4830 | 39.09 | 737/1794 | 41.08 | 823/2012 | 40.90 |
| BodgeIt | 0 | RXSS | 2 | 4 | 20927/53706 | 38.97 | 20648/53130 | 38.86 | 3918/8761 | 44.72 | 4551/9760 | 46.63 |
| BodgeIt | 0 | SXSS | 3 | 2 | 32/345 | 9.28 | 32/345 | 9.28 | 37/250 | 14.80 | 38/252 | 15.08 |
| BodgeIt | 0 | SXSS | 3 | 3 | 561/4875 | 11.51 | 551/4830 | 11.41 | 257/1794 | 14.33 | 273/2012 | 13.57 |
| BodgeIt | 0 | SXSS | 3 | 4 | 6052/53706 | 11.27 | 6024/53130 | 11.34 | 1504/8761 | 17.17 | 1590/9760 | 16.29 |
| BodgeIt | 0 | SXSS | 4 | 2 | 308/345 | 89.28 | 305/345 | 88.41 | 217/250 | 86.80 | 215/252 | 85.32 |
| BodgeIt | 0 | SXSS | 4 | 3 | 4434/4875 | 90.95 | 4407/4830 | 91.24 | 1558/1794 | 86.85 | 1745/2012 | 86.73 |
| BodgeIt | 0 | SXSS | 4 | 4 | 42898/53706 | 79.88 | 42378/53130 | 79.76 | 7676/8761 | 87.62 | 8618/9760 | 88.30 |
| Gruyere | 0 | RXSS | 1 | 2 | 122/345 | 35.36 | 122/345 | 35.36 | 89/250 | 35.60 | 92/252 | 36.51 |
| Gruyere | 0 | RXSS | 1 | 3 | 1744/4875 | 35.77 | 1755/4830 | 36.33 | 671/1794 | 37.40 | 758/2012 | 37.67 |
| Gruyere | 0 | RXSS | 1 | 4 | 19381/53706 | 36.09 | 19223/53130 | 36.18 | 3303/8761 | 37.70 | 3566/9760 | 36.54 |
| Gruyere | 0 | SXSS | 2 | 2 | 23/345 | 6.67 | 23/345 | 6.67 | 17/250 | 6.80 | 18/252 | 7.14 |
| Gruyere | 0 | SXSS | 2 | 3 | 327/4875 | 6.71 | 322/4830 | 6.67 | 118/1794 | 6.58 | 136/2012 | 6.76 |
| Gruyere | 0 | SXSS | 2 | 4 | 3587/53706 | 6.68 | 3542/53130 | 6.67 | 610/8761 | 6.96 | 749/9760 | 7.67 |
| Webgoat | 0 | RXSS | 2 | 2 | 198/345 | 57.39 | 201/345 | 58.26 | 145/250 | 58.00 | 153/252 | 60.71 |
| Webgoat | 0 | RXSS | 2 | 3 | 2842/4875 | 58.30 | 2842/4830 | 58.84 | 1073/1794 | 59.81 | 1207/2012 | 59.99 |
| Webgoat | 0 | RXSS | 2 | 4 | 31441/53706 | 58.54 | 31120/53130 | 58.57 | 5366/8761 | 61.25 | 6084/9760 | 62.34 |
| DVWA | 0 | RXSS | 1 | 2 | 175/345 | 50.72 | 178/345 | 51.59 | 128/250 | 51.2 | 134/252 | 53.17 |
| DVWA | 0 | RXSS | 1 | 3 | 2517/4875 | 51.63 | 2520/4830 | 52.17 | 954/1794 | 53.18 | 1081/2012 | 53.73 |
| DVWA | 0 | RXSS | 1 | 4 | 27864/53706 | 51.88 | 27578/53130 | 51.91 | 4755/8761 | 54.27 | 5345/9760 | 54.76 |
| DVWA | 0 | SXSS | 2 | 2 | 91/345 | 26.38 | 92/345 | 26.67 | 88/250 | 35.20 | 84/252 | 33.33 |
| DVWA | 0 | SXSS | 2 | 3 | 1303/4875 | 26.73 | 1302/4830 | 26.96 | 537/1794 | 29.93 | 616/2012 | 30.62 |
| DVWA | 0 | SXSS | 2 | 4 | 14068/53706 | 26.19 | 13928/53130 | 26.21 | 2276/8761 | 25.98 | 2825/9760 | 28.96 |
| DVWA | 1 | RXSS | 1 | 2 | 106/345 | 30.72 | 109/345 | 31.59 | 80/250 | 32.00 | 86/252 | 34.13 |
| DVWA | 1 | RXSS | 1 | 3 | 1548/4875 | 31.75 | 1554/4830 | 32.17 | 613/1794 | 34.17 | 692/2012 | 34.39 |
| DVWA | 1 | RXSS | 1 | 4 | 17173/53706 | 31.98 | 16952/53130 | 31.91 | 3285/8761 | 37.50 | 3787/9760 | 38.80 |
| DVWA | 1 | SXSS | 2 | 2 | 0/345 | 0.00 | 0/345 | 0.00 | 0/250 | 0.00 | 0/252 | 0.00 |
| DVWA | 1 | SXSS | 2 | 3 | 0/4875 | 0.00 | 0/4830 | 0.00 | 0/1794 | 0.00 | 0/2012 | 0.00 |
| DVWA | 1 | SXSS | 2 | 4 | 0/53706 | 0.00 | 0/53130 | 0.00 | 0/8761 | 0.00 | 0/9760 | 0.00 |

vectors) and its respective percentage for both IPOG and IPO-F algorithms. Further, the table gives the corresponding results for constrained values in each case.

In DVWA, higher interaction strengths caused higher exploitation rates when using the constrained test set. This was the case with test sets from both algorithms. When the same element in the SUT was tested against reflected XSS, the program obtained 32% for $t = 2$ and 37.50% for the $t = 4$ for IPOG. The other algorithm gave for the same elements results around 34.13% and 38.80%. A slight increase for constrained vectors was also reported in Webgoat with IPOG, where for $t = 2$ the output has been 58% and 61.25% for the highest strength. A slight increase of 1.63% was also reported when using IPOG-F for the same SUT. When Mutillidae was tested with unconstrained vectors, we achieved 29.57% for $t = 2$ and corresponding 32.24% for $t = 4$.

On the other hand, when comparing the testing outcome from both algorithms regarding the test sets, the most evident difference is obtained in Mutillidae. In this SUT the exploitation rate differs most between constrained values and their counterpart. While the overall results for the first tested element result around 32% for all interaction strengths in IPOG, for the same algorithm we achieved much higher results with constrained vectors. In this case, the program reported respectively 46.60% and 48.02% in favor for IPOG-F. Similar results were observed after testing another element of this SUT. However, the program was not able to trigger any vulnerability for another element. The assumption is that this part uses defense mechanisms that reject parts of the vectors. Also, we obtained equal results when the SUT was tested against the same test sets but on a higher difficulty level. In this case, additional attack prevention mechanisms were activated inside the SUT. However, vectors that were before able to succeed in triggering XSS, also caused the same effect in the upgraded SUT.

In DVWA the program calculated 31.75% for IPOG for $t = 3$, while achieving 34.17% for the same algorithm when using constrained values. When tested with $t = 4$, results varied even more with respectively 31.91% and 38.80% for IPOG-F. In BodgeIt the results were 79.88% for unconstrained vectors in IPOG but 87.62% for the other test set when tested against stored XSS. Similar results were obtained while testing for reflected XSS.

We will discuss the test results for both algorithms for Webgoat. This SUT was tested just for reflected XSS against one element. In this case, an exploitation rate of about 57.39% was observed for the lowest interaction strength in IPOG. However, the biggest difference is calculated when using constrained vectors. Here we obtained a slight increase in the exploitation rate with IPOG-F when tested against $t = 2$. Moreover, we observe a result of about 60.71% compared to the same test set from the other algorithm, where we obtained only 58%. A slightly smaller difference was achieved for the unconstrained counterpart. However, here the exploitation result was higher for IPOG-F for every interaction strength. A better result is also confirmed for $t = 4$ where IPOG got 61.25% but IPOG-F succeeded with additional 1.09%.

To summarize the evaluation results of this section, in the majority of the input fields of the SUTs, we witnessed an increase in the exploitation rate when changing `G` with `G_c` and it is clear that using constraints results in better quality attack vectors. In addition, IPOG-F gives overall better results than IPOG and we recommend the usage of the first algorithm when generating tests for web security testing in terms of exploitation rate. Last but not least, we confirm the fundamental rule of combinatorial testing; testing with higher interaction strength makes likely to reveal more errors. In our context, we interpret and confirm this rule in terms of exploitation rate, i.e. increasing the interaction strength implies higher exploitation rates when these are tested for XSS vulnerabilities.

### *B. Combinatorial Coverage Measurement for Web Security Testing*

In this section, we computed the simple t-way combination coverage of passing tests derived from our experiments for Gruyere, DVWA and Webgoat in the CCM tool. In particular, we took into account test suites that were generated only via `G` as we encountered some problems parsing the constraints of `G_c` in CCM. The CCM tool offers a user-friendly GUI interface to perform the experiments and has the functionality to print combination coverage charts. We would like to note that in contrast to prior applications of CCM for evaluating combination coverage in large test suites that are not necessarily designed using covering arrays (e.g. [17]), in our case the passing tests (successful XSS exploits) come from test suites that are produced through combinatorial testing. This feature plays an important role in the evaluation of our passing tests.

In Table II. we give information about the combinatorial interaction strength (Str.) of the test suites that the passing tests were originated, the SUT (App), the simple $t$-way combination coverage for $t \in \{2, 3, 4\}$ we want to measure and its respective percentage for the passing tests that their original test suites were generated with the IPOG and IPOG-F algorithms. In addition, we list the number of $t$-way combinations in each case denoted by $C(11, t)$. Recall that, combination coverage is measured as the proportion of fully covered $t$-way combinations of given $k$ parameters in a test suite.

Moreover, in Figures 1, 2 and 3 we give a visualization of the combination coverage (in Y axis) with the percentage of combinations reaching a particular coverage (X axis) for DVWA, Gruyere and Webgoat, respectively.

From the evaluation results of this table, we see that for DVWA and Gruyere applications when we measure the simple 2-way combination coverage in the passing tests of the test suites that were generated with interaction strength $t = 3$ and $t = 4$ in both IPOG and IPOG-F algorithms, this selection of passing tests forms a covering array of strength 2 as all 2-way combinations of the 11 variables of our attack grammar are fully covered. Usually, large test suites naturally cover a high percentage of $t$-way combinations but we are unaware of a case study for web security testing where the passing tests are a covering array. However, after some post-processing of these results we want to note that the cardinality of each one of the 11 variables is in some cases smaller than the ones we presented in Section II-B. This implies that some variable values in our attack grammar do not contribute in revealing XSS vulnerabilities. This could lead to a method to reverse engineer the structure of successful vectors in order to achieve better results in terms of exploitation rate and will be explored further in future research.

For example, when testing Webgoat with the test suites generated via IPOG-F algorithm, for interaction strength $t \in \{2, 3, 4\}$ we see that **JSO**(1)=`<scr<script>ipt>` does not appear in any of the variable-value configurations of the $t$-way combinations of the parameters of our attack grammar in passing tests. As a first step to refine the attack grammar this would mean to remove this value from the parameter JSO. At this point, we want to mention that is also related to the filter mechanisms of the Webgoat application but since we are interested in successful attack vectors as a discrete structure in this paper we will not elaborate further on this security testing perspective.

Moreover, from the evaluation results in Table II. we see that IPOG-F again gives better results than IPOG algorithm. Another point we would like to mention to conclude this evaluation is that when taking into account the measurement results for simple 2-way combination coverage in the passing tests of the test suites generated with interaction strength $t = 2$, we see some “hidden” variable-value configurations of 3-way and 4-way combinations appearing. This implies that such combinations will produce successful attack vectors w.r.t. XSS exploits also for higher interaction strengths.

## VI. Conclusion and Future Work

In this work we demonstrated the results of our effort to improve the previous works from the authors on security

TABLE II. EVALUATION RESULTS FOR MEASURING COMBINATION COVERAGE PER SUT WITH INCREASING STRENGTH.

| | | | IPOG | | | | | | IPOG-F | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| App | | C(11,t) | t=2 | | t=3 | | t=4 | | t=2 | | t=3 | | t=4 | |
| DVWA | 2-way | 55 | 2477/2922 | 84.77% | 2922/2922 | 100.00% | 2922/2922 | 100.00% | 2507/2922 | 85.80% | 2922/2922 | 100.00% | 2922/2922 | 100.00% |
| | 3-way | 165 | 16676/57812 | 28.85% | 54066/57812 | 93.52% | 57803/57812 | 99.98% | 18256/57812 | 31.58% | 54223/57812 | 93.79% | 57803/57812 | 99.98% |
| | 4-way | 330 | 46388/716350 | 6.48% | 340821/716350 | 47.58% | 700147/716350 | 97.74% | 51826/716350 | 7.23% | 349377/716350 | 48.77% | 700739/716350 | 97.82% |
| Gruyere | 2-way | 55 | 2076/2772 | 74.89% | 2771/2772 | 99.96% | 2771/2772 | 99.96% | 2106/2772 | 75.97% | 2771/2772 | 99.96% | 2771/2772 | 99.96% |
| | 3-way | 165 | 12469/53168 | 23.45% | 46270/53168 | 87.03% | 53086/53168 | 99.85% | 13637/53168 | 25.65% | 46842/53168 | 88.10% | 53086/53168 | 99.85% |
| | 4-way | 330 | 32522/637878 | 5.10% | 262873/637878 | 41.21% | 601575/637878 | 94.31% | 36445/637878 | 5.71% | 271808/637878 | 42.61% | 602139/637878 | 94.40% |
| Webgoat | 2-way | 55 | 2610/2997 | 87.09% | 2997/2997 | 100.00% | 2997/2997 | 100.00% | 2661/2997 | 88.79% | 2997/2997 | 100.00% | 2997/2997 | 100.00% |
| | 3-way | 165 | 18368/60134 | 30.55% | 56710/60134 | 94.31% | 60125/60134 | 99.99% | 19998/60134 | 33.26% | 57200/60134 | 95.12% | 60125/60134 | 99.99% |
| | 4-way | 330 | 52355/755586 | 6.93% | 368455/755586 | 48.76% | 742208/755586 | 98.23% | 57843/755586 | 7.66% | 379774/755586 | 50.26% | 743397/755586 | 98.39% |

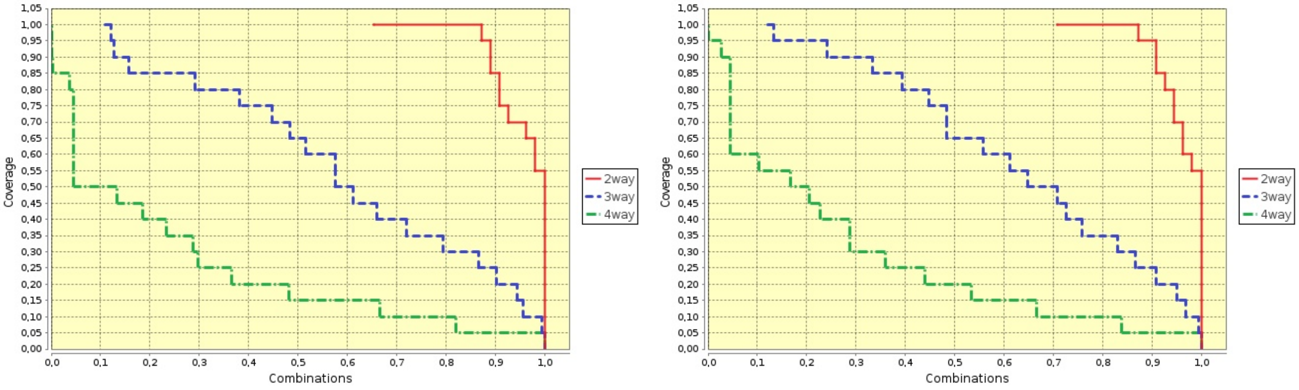

Fig. 1. Comparison of combination coverage measurement for passing tests in DVWA (inp_id 1, DL 0) when their respective test suites are generated in IPOG (left) and IPOG-F (right) with interaction strength $t = 2$.

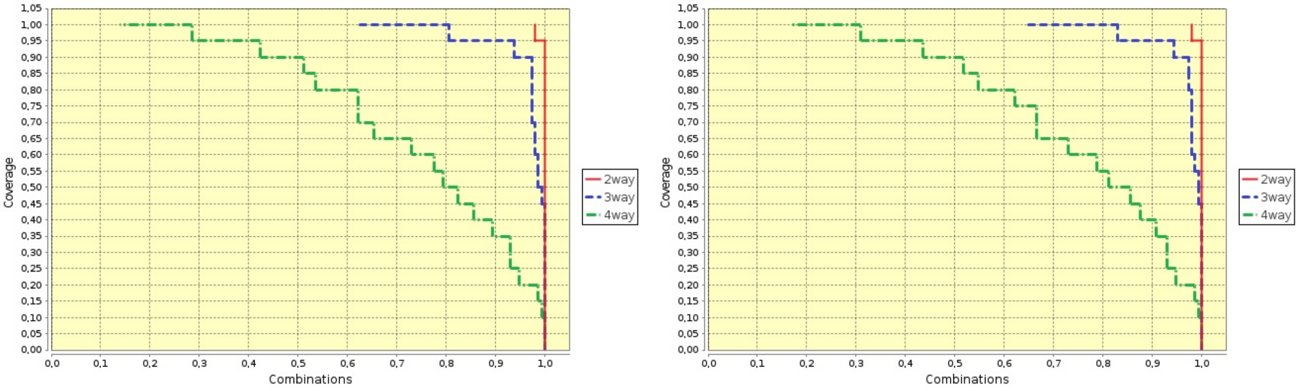

Fig. 2. Comparison of combination coverage measurement for passing tests in Gruyere (inp_id 1, DL 0) when their respective test suites are generated in IPOG (left) and IPOG-F (right) with interaction strength $t = 3$.

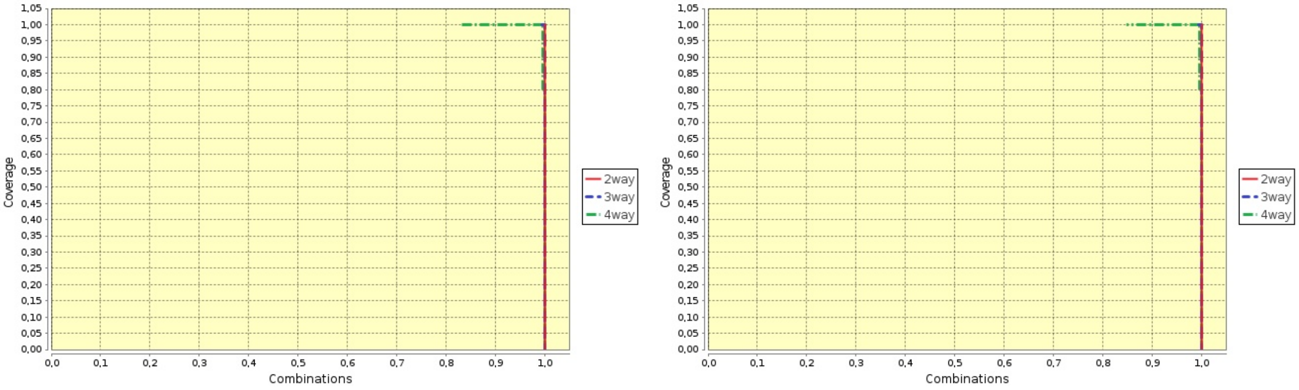

Fig. 3. Comparison of combination coverage measurement for passing tests in Webgoat (inp_id 2, DL 0) when their respective test suites are generated in IPOG (left) and IPOG-F (right) with interaction strength $t = 4$.

testing of web applications by studying the latter problem from a combinatorial testing perspective. We made an extensive comparison for two of the main algorithms of the IPO-family, namely IPOG and IPOG-F, when these are used to generate test suites to be used against web applications for revealing security vulnerabilities. In this regard, our findings from this experience report indicate that IPOG-F performs better in the sense that it generates better quality attack vectors. We would also like to highlight that testing with combinatorial attack grammars with increasing interaction strength results in higher exploitation rates in most of the cases. In addition, setting constraints inside the input model results in significantly improved attack vectors generated from both algorithms.

Our evaluation results are further supported by measuring the combination coverage of successful attack vectors in some our test suites. These computations confirm that the revealed security leaks in web applications are mainly due to a handful of parameters but also gave some insight on how to reverse engineer attack grammars for web security testing. We plan to explore this promising approach, in detail, in future work.

## Acknowledgment

The research presented in the paper has been funded in part by the COMET K1 Program by the Austrian Research Promotion Agency (FFG).

## References

[1] J. Williams and D. Wichers, "OWASP Top 10 2013," 2013, https://www.owasp.org/index.php/Top_10_2013.

[2] "sqlmap," http://sqlmap.org/, accessed: 2014-10-20.

[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] 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.

[5] J. Bozic, B. Garn, I. Kapsalis, D. E. Simos, S. Winkler, and F. Wotawa, "Attack pattern-based combinatorial testing with constraints for web security testing," 2015, submitted for publication.

[6] 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.

[7] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Software Engineering*, Innsbruck, Austria, Feb. 2007.

[8] 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.

[9] I. Dominguez Mendoza, D. Kuhn, R. Kacker, and Y. Lei, "CCM: A tool for measuring combinatorial coverage of system state space," in *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, 2013, pp. 291–291.

[10] M. Brcic and D. Kalpic, "Combinatorial testing in software projects," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 1508–1513.

[11] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[12] M. Mehta and R. Philip, "Applications of combinatorial testing methods for breakthrough results in software testing," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 348–351.

[13] L. Ghandehari, M. Bourazjany, Y. Lei, R. Kacker, and D. Kuhn, "Applying combinatorial testing to the siemens suite," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, 2013, pp. 362–371.

[14] M. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn, "Combinatorial testing of acts: A case study," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 591–600.

[15] D. Richard Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, ser. SEW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 153–158.

[16] A. M. Cunningham Jr., J. Hagar, and R. J. Holman, "A system analysis study comparing reverse engineered combinatorial testing to expert judgment," 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. 630–635.

[17] J. Maximoff, M. Trela, D. Kuhn, and R. Kacker, "A method for analyzing system state-space coverage within a t-wise testing framework," in *Systems Conference, 2010 4th Annual IEEE*, 2010, pp. 598–603.

[18] Z. Zhang, X. Liu, and J. Zhang, "Combinatorial testing on id3v2 tags of mp3 files," 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. 587–590.

[19] E. Puoskari, T. E. J. Vos, N. Condori-Fernandez, and P. M. Kruse, "Evaluating applicability of combinatorial testing in an industrial environment: A case study," in *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, ser. JAMAICA 2013. New York, NY, USA: ACM, 2013, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2489280.2489287

[20] 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.

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

[22] C. Phillips and L. Swiler, "A graph-based system for network vulnerability analysis," in *ACM New Security Paradigms Workshop*, 1998, pp. 71–79.

[23] 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.

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

[25] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[26] 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.

[27] 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.

[28] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection," in *CODASPY*. ACM, 2014, pp. 37–48.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG-IPOG-D: Efficient test generation for multi-way combinatorial testing," *Softw. Test. Verif. Reliab.*, vol. 18, no. 3, pp. 125–148, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1002/stvr.v18:3

[34] M. Forbes, J. Lawrence, Y. Lei, R. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, pp. 287–297, 2008.

[35] Y. Lei and K.-C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, ser. HASE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 254–261. [Online]. Available: http://dl.acm.org/citation.cfm?id=645432.652389

[36] J. Zhang, Z. Zhang, and F. Ma, "The IPO family," in *Automatic Generation of Combinatorial Test Data*, ser. SpringerBriefs in Computer Science. Springer Berlin Heidelberg, 2014, pp. 41–49. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43429-1_4

[37] 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.

[38] 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.

[39] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Common patterns in combinatorial models," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 624–629.

[40] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 199–209.

[41] 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.

[42] 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.