A Fault-Driven Combinatorial Process for Model Evolution in XSS Vulnerability Detection

Bernhard Garn[1], Marco Radavelli[2], Angelo Gargantini[2], Manuel Leithner[1], and Dimitris E. Simos[1]

[1] SBA Research – 1040 Vienna, Austria
{bgarn,mleithner,dsimos}@sba-research.org
[2] University of Bergamo – Bergamo, Italy
{marco.radavelli,angelo.gargantini}@unibg.it

Abstract. We consider the case where a knowledge base consists of interactions among parameter values in an input parameter model for web application security testing. The input model gives rise to attack strings to be used for exploiting XSS vulnerabilities, a critical threat towards the security of web applications. Testing results are then annotated with a vulnerability triggering or non-triggering classification, and such security knowledge findings are added back to the knowledge base, making the resulting attack capabilities superior for newly requested input models. We present our approach as an iterative process that evolves an input model for security testing. Empirical evaluation on six real-world web application shows that the process effectively evolves a knowledge base for XSS vulnerability detection, achieving on average 78.8% accuracy.

Keywords: Combinatorial testing · XSS vulnerability · security testing · model evolution

1 Introduction

Computer users of today often interact via web-applications with services offered by various parties. This new way of connecting the client and server side with each other has brought about a multitude of novel security challenges, both for the client and the server [3]. One major threat to web applications is posed by Cross-Site Scripting (XSS), which continues to be included in the OWASP Top 10 most critical web application security risks [5]. Security testing is a vital and expensive part of the software development lifecycle. An effective testing technique applied to security testing is *Combinatorial Testing* (CT), for the capability of detecting failures and fail conditions with a small amount of tests that need to be executed compared to the whole input space [14]. Given a discrete finite model of the *system under test* (SUT), made of parameters with a finite list of possible values, called *input parameter model* (IPM), and given an interaction strength t, CT creates a test suite guaranteeing the appearance of all t-way interactions of parameter values, for any selection of t parameters [11]. In the case for testing for exploiting XSS vulnerabilities, the IPM specifies an attack grammar and is also called *(abstract) attack model*. The aim of this paper is to present a way to

```
Model wavsep_xss

Parameters:
  JSO: { P1 P2 P3 P4 P5 P6 P7 P8}            PAY: { P1 P2 P3 P4 P5 }
  INT: { P1 P2 P3 P4 P5 P6 P7 P8 P9 P10}     EVH: { P1 P2 P3 }
  PAS: { P1 P2 P3 P4 P5 P6 P7 P8 P9 P10}     WS: Boolean
  JSE: { P1 P2 P3 P4 P5 P6 P7 P8 P9}

Constraints:
  # ! JSE==JSE.P2 #     # ! JSE==JSE.P3 #          # ! JSE==JSE.P4 #
  # ! INT==INT.P9 #     # ! (JSO==JSO.P2 && WS==false && PAS==PAS.P7) #
   ...
```

Fig. 1: Knowledge base K_3 for NavigateCMS: abstract attack model (initially it had no constraints), with detected XSS vulnerability constraints, in CTWedge

evolve knowledge bases for security testing. In our approach, the evolution of a knowledge base consists in the integration of learned constraints, using BEN [9], into the IPM. In particular, the contribution of this paper consists in an automated technique to detect all the conditions under which vulnerabilities are triggered, by using combinatorial testing. Evaluation shows that the process is able to evolve the knowledge base to achieve, on average over all the benchmarks, 78.8% accuracy, in 14 minutes computation time.

The rest of the paper is structured as follows. In Sect. 2 we give basic definitions, in Sect. 3 we discuss our proposed process, and Sect. 4 presents the results of our case study experiments. We provide a brief overview over related work in Sect. 5, and we conclude the paper with further research directions in Sect 6.

2 Preliminaries

In the course of *combinatorial security testing* (cf. [14]), attack models have appeared in the form of a BNF grammar, e.g. [2], [13]. In this paper, we will follow this established terminology for designing XSS attack models to be used in conjunction with combinatorial methods. We denote with K_i a knowledge base at time i, encoded as an abstract attack model (IPM, see Fig. 1). Given an IPM, an abstract test case f is a particular assignment of values for its parameters. An abstract test suite is used to derive a concrete test suite, where abstract test cases are being translated into concrete XSS attack strings via a translation function τ. For example, given the following abstract test case:

$$(\mathsf{JSO} = 2, \mathsf{WS} = 1, \mathsf{INT} = 3, \mathsf{EVH} = 2, \mathsf{PAY} = 2, \mathsf{PAS} = 5, \mathsf{JSE} = 7)$$

for each parameter, the respective integer value corresponds to a concrete parameter value (i.e., a string), and the translated concrete test case is obtained by concatenating all these strings together in the order given by the IPM:

```
<script> onError= alert(1) ') '\>
```

The resulting string can be submitted against the SUT, and a boolean function `orac` decides if the outcome of the execution of the translated test case

$\tau(f)$ against the SUT triggers an XSS vulnerability or not. We denote also the function $eval(f) := oracle(\tau(f))$, that is *true* if the test case f triggered an XSS vulnerability (i.e., f is a *vulnerability triggering* test case). The generated test vectors aim at producing valid JavaScript code when these are executed against SUTs. A description of parameters that appear in the attack model is mentioned in [2, 8, 13]. At any time point i, the knowledge base K_i may be used to create test cases, and to classify an abstract test case as either vulnerability-triggering or not, depending on whether the *constraints* are satisfied. This capability of the knowledge base is denoted as a *model* function, which takes as input an abstract test case, and gives as output a "best guess" that may or may not be correct w.r.t. the actual result of the function *eval*. The attack model initially contains only combinatorial parameters and no constraints. During the process, the knowledge base is enriched by the conditions used to identify the vulnerabilities.

To put this problem into a formal setting, a *knowledge base fault* occurs when a test f is classified as non-vulnerable in the model ($\neg model(f)$), despite it actually triggers a vulnerability in the SUT ($eval(f)$) (**False Negative**: it entails a loss of potentially valuable information for fixing the vulnerability); or when a test is being marked as vulnerability-triggering in the model ($model(f)$), despite it does not trigger a vulnerability ($\neg eval(f)$) (**False Positive**: it triggers a false alarm, and the programmers may consequently waste effort in fixing pieces of code that did not trigger any vulnerability). When such a discrepancy is fixed by updating the model function, we say that the knowledge base evolves.

Since in our experiments it yielded better results, we decided to consider the convention for which the initial model function considers any test to be non-vulnerability triggering; and during the process constraints are added in order to identify and subsequently exclude all the tests that do not trigger any vulnerability. We call this convention *pessimistic* approach, in contrast with the *optimistic* one in which initially any test is vulnerability-triggering, and constraints are added to isolate the tests that actually trigger some vulnerability.

Let us complete some notation for combinatorial analysis. A combination c is an assignment (i.e. configuration) on a subset $Dom(c)$ of all the possible parameters P in the attack model, such that $Dom(c) \subseteq P$. We call *size* of the combination the cardinality of $Dom(c)$. A combination c identifies a set of tests: c *represents* a test f if all the parameters in c are also present in f, associated to the same values. Formally, $c \subseteq f : \forall p \in Dom(c), f(p) = c(p)$. A combination c is *suspicious* in a test set $F \subseteq \Gamma$ if c represents *only failed tests in* F. Formally, $\forall f \in F : c \subseteq f \rightarrow model(f) \neq eval(f)$. For the purposes of this work, we assume that the constraints are in conjunctive relation among each other.

3 Process for Model Evolution

Fig. 2 shows an overview of our process to automatically evolve an abstract attack model initialized without constraints, to detect conditions that trigger XSS vulnerabilities. The process proceeds according to the following steps:

1. From a defined initial interaction strength t, derive a t-way test suite.

2. Mark the test cases as failing or passing according to the current model and the evaluator. If all the tests pass and Th_t is not yet reached, increment the

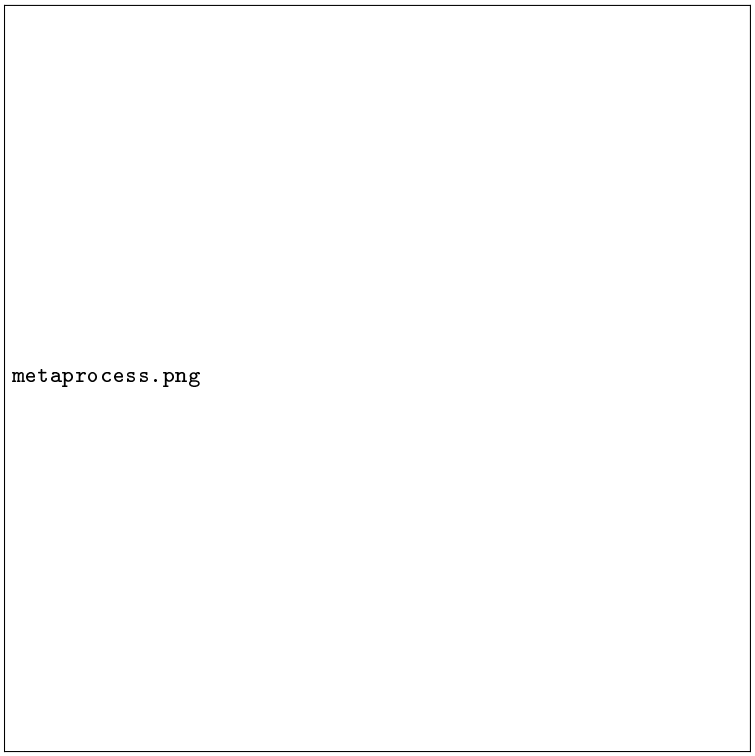

Fig. 2: Condition detection meta-process

strength $t \leftarrow t+1$, and go to point 1. Otherwise, evaluate the tests. Internally, the evaluator executes the translation function τ to obtain the concrete test string to insert in the *reflection* URL to query the SUT. PhantomJS[3] then analyzes the HTML code returned by the SUT for a specific *target function* (the `alert()` function in our case[4]) that was included as payload. If any of the target functions were executed, the injection was successful; if the page loads normally and does not produce any errors, the injection is deemed unsuccessful. Lastly, if JavaScript errors are observed on the page, it is likely that some content was injected, but not in a form that constitutes a usable injection (thus resulting in incorrect syntax). Our current approach regards these test cases as failing, but future evolutionary approaches might take advantage of this particular classification.

3. Pass the evaluated test suite to BEN [9] to derive suspicious combinations, together with their *suspiciousness* level. We call BEN multiple times specifying

[3] PhantomJS (http://phantomjs.org/) is a headless browser environment enabling introspection of events such as network requests, document edits and JavaScript errors.

[4] in theory, any valid JavaScript functions that will not be called during the normal operation of the SUT can be chosen instead.

the size t_{BEN} of the suspicious combinations to detect[5], from 1 to Th_{BEN}. BEN may also ask for a few additional tests (up to 10 tests at a time: *inner BEN cycles*) to reduce the amount of suspicious combinations and improve the accuracy of the computed *suspiciousness* levels.

4. The suspicious combinations from BEN are then translated into a set of constraints for the current knowledge base K_i, by negating the corresponding boolean expression (obtained by putting the assignments in conjunction) of every combination whose *suspiciousness* value is above the threshold Th_S. Due to low accuracy in the detected suspicious combinations, we noticed that K_i often results to be a contradiction, which is normally not the case in real-world systems.Therefore, we *post-process* the constraints by computing the *unsat-cores* of K_i and removing all such clauses starting from the *least-suspicious* constraints (according to BEN), until K_i is not a contradiction any longer. The process eventually quits if either the user is *satisfied* with the quality of K_i[6], or the threshold Th_t is reached. Otherwise, increase t and go to point 1.

4 Experiments

The process has been implemented in Java using `CTWedge` [7] to represent and update attack models, `ACTS` [16] to generate combinatorial test suites of a defined strength, and `BEN` [9] as a tool to detect suspicious combinations and compute suspiciousness. Experiments were executed on a PC with Intel i7 3.40GHz processor and 16 GB RAM. We run the process on six real web-applications: four are part of the WAVSEP[7] project, and two are open source content management systems: MiniCMS and NavigateCMS. Each SUT receives over HTTP one GET parameter which is rejected on the page in different contexts, and might optionally be altered by a specific sanitization function. Tab. 1 shows, for each SUT, the respective *vulnerability ratio*, i.e., the ratio of tests that triggered an XSS vulnerability ($eval(f)$) out of the total number of tests executed, that, given the practical infeasibility of the exhaustive test suite, we compute on all the tests generated up to strength t=5 (42830 tests).[8]

To assess the quality of the evolved knowledge base from our method, we use the typical metrics of information retrieval: in particular, *precision* ($\frac{TP}{TP+FP}$), and *recall* ($\frac{TP}{TP+FN}$) give a measure of how the process isolates true positives, *accuracy* gives an overall ratio of correctly classified tests, and the F_1 score is considered to be a good candidate synthesis index of the inferred model's quality. For the experiments, we set the parameters of the process as follows:

- $Th_{BEN} = 3$. We limited the size of detected suspicious combinations as the BEN process becomes too slow when computing suspiciousness of the too many combinations of size 4 (or larger) on these attack models.

[5] note that t_{BEN} is different from the strength t for generating the initial test suite

[6] in this case, we believe that the suspiciousness average and standard deviation could be useful indicators of the F_1 score that the currently inferred model may have

[7] WAVSEP: Web Application Vulnerability Scanner Evaluation Project, `https://github.com/sectooladdict/wavsep`.

[8] The tests suites were generated using the IpoF algorithm, implemented in ACTS

Table 1: XSS reflection sites on WAVSEP benchmarks

SUT ID	SUT name	Reflection site	vulnerability ratio (t=5)
1	Tag2HtmlPageScope	<body>$input</body>	17.08 %
2	Tag2TagStructure	<input type="text" value="$input">	4.06 %
3	Event2TagScope	<img src="$input">	4.63 %
4	Event2DoubleQuotePropertyScope	<img src="$input">	3.45 %
5	MiniCMS	/mc-admin/page.php?date=$input	80.2 %
6	NavigateCMS	/navigate.php?fid=$input	60.0 %

Table 2: Quality metrics for the inferred models ($Th_{BEN} = 3$, and $Th_S = 0$)

	sut	time (s)	constraints	suspiciousness avg. ± s.d.	accuracy	precision	recall (TPR)	specificity (TNR)	F_1 score
Th_t=4	1	246	146	0.254 ± 0.0288	72.6	32.5	55.7	76.1	41.0
	2	141	38	0.362 ± 0.00753	93.3	32.6	59.5	94.8	42.1
	3	1437	794	0.137 ± 0.0456	87.4	15.7	39.7	89.7	22.5
	4	1088	551	0.136 ± 0.0483	89.0	13.7	42.3	90.6	20.7
	5	1445	623	0.342 ± 0.0287	78.3	87.0	85.8	48.3	86.4
	6	679	80	0.344 ± 0.0147	52.0	66.6	40.0	70.0	50.0
	avg	839	372	0.263 ± 0.0289	78.8	41.4	53.8	78.3	43.8
Th_t=3	1	9.3	644	0.196 ± 0.0423	41.4	19.8	79.8	33.5	31.8
	2	4.2	164	0.224 ± 0.0874	78.9	14.2	83	78.7	24.2
	3	28.5	764	0.207 ± 0.0628	75.4	12.9	75.7	75.3	22
	4	9.3	123	0.125 ± 0.0379	72.3	8.53	73.7	72.2	15.3
	5	58.5	1340	0.318 ± 0.0268	80.1	80.2	99.9	0.306	89
	6	37.6	2460	0.296 ± 0.0245	62.2	61.8	97.1	9.91	75.5
	avg	24.6	915	0.228 ± 0.0470	68.4	32.9	84.9	45.0	43.0

- t, the initial strength of test suite, is set to 2.
- $Th_t = 4$. We limit the maximum strength of the initial test suite since even t=5 (about 36000 tests) would make the BEN process too slow.
- $Th_S = 0$, as we want all the suspicious combinations to be considered.

Test suites for interaction strengths $t \in \{3, 4\}$ had 900 and 7200 test cases, respectively. For each SUT, Tab. 2 reports the number of constraints included in the inferred model, the average suspiciousness with its standard deviation, and the accuracy, precision, recall, specificity, and F_1 score of the final model, computed over all tests up to strength 5, as for the *vulnerability ratio* in Tab. 1.

RQ1: *What is the quality of the model obtained by the approach?* We observe that the inferred model achieves an average accuracy of 78.8%, with a maximum of 93.3%. Precision has an average of 41.4%, ranging from 13.7% to 87%, and recall (54% on average) is higher than precision. F_1 score is on average 43.8%, with a maximum of 86.4% on SUT5. With relatively few tests (t=4 out of 7 parameters), the final model is of good quality, but not completely accurate. We can also observe that F_1 is proportional to the vulnerability ratio of the SUT.

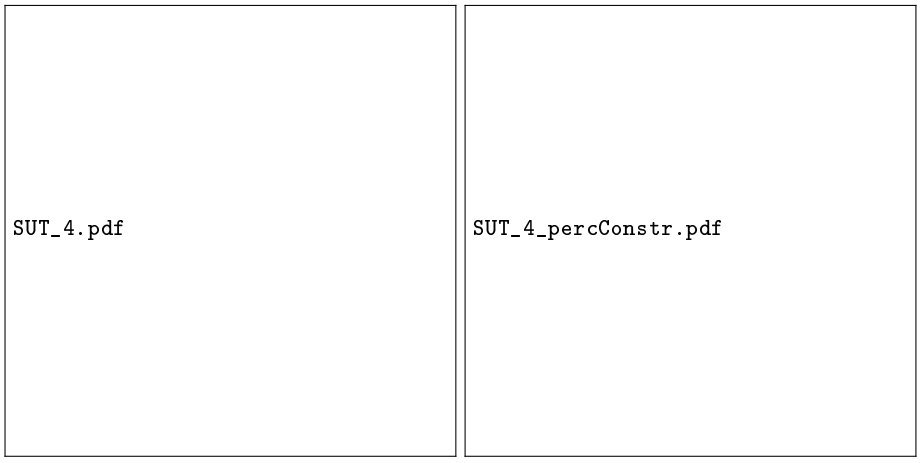

Fig. 3: Achieved F_1 score of final model by varying Th_S, when $Th_t = 4$

RQ2: *How does the quality of the inferred model vary depending on Th_t?* By increasing Th_t from 3 to 4, the time taken, as expected, increases, while the number of constraints, in most cases, decreases, meaning that with more tests our process is able to describe the vulnerability conditions with fewer constraints. On average, although recall decreases, both precision, accuracy and specificity increase, and F_1 slightly increases. This means that the classification improves.

RQ3: *Which is the computational effort of the proposed process?* Tab. 2 also reports the total execution time, excluding the actual test execution, as test results are cached, except for the few tests (30 at most) that BEN may ask during the process. For the first two SUTs, the process takes less than 250 seconds to complete, but up to 24 minutes are needed for SUT5 with $Th_t = 4$. Most of the computation time is used internally by BEN; by limiting to 3 the strength of the initial test suite, the total time is always below 1 minute for every SUT.

RQ4: *How does variations of Th_S affect model quality?* The highest F_1 score is achieved with low values of Th_S (see Fig. 3), except for SUT3 and SUT4, for which a $\overline{Th_S} \simeq 0.13$ achieves the maximum F_1 score. However, we can notice that at least around 25% of the constraints (starting from the least suspicious ones) can be removed with negligible impact on the final F_1 score.

5 Related Work

XSS vulnerability detection is not a novel topic in computer science research. Duchene et al. [4] used model based testing and fuzzing to discover XSS vulnerabilities; Melicher et al. [12] proposed improvements on using the DOM model to generate and detect XSS attacks; Simos et al. [13] proposed a combinatorial approach to find attack vectors that trigger XSS vulnerabilities; Jia et al. [10] used machine learning and hyper-heuristic search to improve combinatorial tests; Temple et al. [15] proposed a machine-learning approach to infer

constraints among parameters that, although not sound, achieves high precision (about 90%) and recall (80%). Although these works use model-based testing,the usage of combinatorial testing for XSS vulnerability detection to *classify* vulnerabilities based on the input and describe *completely* the vulnerability space of a part of a web application, evolving a *knowledge base*, is the main novelty of our approach. The first phase of BEN [9] as a failure-inducing combination detection and ranking tool has been used by Gargantini et al. [6] to repair constraints in combinatorial models, evaluating different test generation policies.

6 Conclusion and Future Work

We presented an automated iterative process based on combinatorial testing to evolve an attack model to include conditions among input parameters that trigger XSS vulnerabilities in web applications. Our approach is based on the notion of suspicious combination, i.e., whose appearance in a test vector would trigger a discrepancy between the *best-guess* of the current model, and the actual outcome when executed against the SUT. Identification of constraints among XSS attack parameters helps to better understand the root cause of an XSS vulnerability and provides insights about how to fix a flawed sanitization function. As future work, we plan to improve the process by reducing the required tests, using information from previous step, and evaluating alternatives to BEN, such as MixTgTe [1]. We believe that this approach can be extended to other security vulnerabilities related to sanitization functions, and to detect discrepancies between a functional system specification and its implementation. Another direction is to further simplify the detected constraints, to reduce them in number and present them to the user in a more readable way.

References

1. P. Arcaini, A. Gargantini, and M. Radavelli. Efficient and guaranteed detection of t-way failure-inducing combinations. In *IEEE International Conference on Software Testing, Verification and Validation ICST Workshops*, 2019.
2. J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, and F. Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. In *IEEE Int. Conf. on Software Quality, Reliability and Security*, 2015.
3. D. Catteddu. Cloud computing: benefits, risks and recommendations for information security. In *Web application security*, pages 17–17. Springer, 2010.
4. F. Duchene, R. Groz, S. Rawat, and J.-L. Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 815–817, 2012.
5. O. Foundation. OWASP Top 10 2017. `https://www.owasp.org/index.php/Top_10-2017_A7-Cross-Site_Scripting_(XSS)`. [Online; accessed 19-April-2018].
6. A. Gargantini, J. Petke, and M. Radavelli. Combinatorial interaction testing for automated constraint repair. In *IEEE Int. Conf. on Software Testing, Verification and Validation ICST Workshops*, pages 239–248, March 2017.
7. A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *IEEE International Conference on Software Testing, Verification and Validation ICST Workshops*, pages 308–317, 2018.
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 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing, pages 16–21. ACM, 2014.

9. L. S. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung, and T. Xie. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, 2018.
10. Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the Int. Conf. on Software Engineering - Volume 1*, ICSE '15, pages 540–550, 2015.
11. D. Kuhn, R. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 2013.
12. W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Proceedings of Network and Distributed System Security Symposium*. Internet Society, 2018.
13. D. E. Simos, K. Kleine, L. S. G. Ghandehari, B. Garn, and Y. Lei. A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS) Vulnerabilities in Web Application Security Testing. In *Testing Software and Systems*. Springer, 2016.
14. D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker. Combinatorial methods in security testing. *IEEE Computer*, 49:40–43, 2016.
15. P. Temple, J. A. Galindo, M. Acher, and J.-M. Jézéquel. Using machine learning to infer constraints for product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 209–218. ACM, 2016.
16. L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. Acts: A combinatorial test generation tool. In *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2013.