Math.Comput.Sci.
https://doi.org/10.1007/s11786-021-00502-7

Mathematics in Computer Science

Factorials Experiments, Covering Arrays, and Combinatorial Testing

Raghu N. Kacker · D. Richard Kuhn · Yu Lei · Dimitris E. Simos

Received: 29 January 2020 / Revised: 9 October 2020 / Accepted: 27 October 2020
© The Author(s), under exclusive licence to Springer Nature Switzerland AG 2021, corrected publication 2021

Abstract In the twenty-first century, our life will increasingly depend on software-based products and complex interconnected systems. Thus, the quality and security of software-based systems is a world-wide concern. Combinatorial testing is a versatile methodology for finding errors (bugs) and vulnerabilities in software-based systems. This paper offers a review of combinatorial testing. Combinatorial testing (CT) methods evolved from investigations which looked like factorial experiments (FE) with pass/fail outcomes. We will discuss the similarities and differences between FE and CT. Use of CT for detecting errors (bugs) in software-based systems has gained significant interest from the international software testing community. Many successful results have been reported from the use of CT to detect software errors in aerospace, automotive, defense, cybersecurity, electronic medical systems, and financial service industries. Now, combinatorial testing methods are being increasingly used to investigate vulnerabilities in software-based systems. Combinatorial testing could be useful in detecting errors and security vulnerabilities in Internet of Things, Autonomous Systems, and Artificially Intelligent Software.

Keywords Design of experiments · Orthogonal arrays · Pairwise-testing · Software testing · Verification and validation of software

Mathematics Subject Classification 00: General and overarching topics · 05: Combinatorics · 68: Computer Science

Disclaimer Identification of any commercial products in this paper does not imply recommendation or endorsement by NIST.

R. N. Kacker · D. R. Kuhn
National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
e-mail: raghu.kacker@nist.gov
e-mail: d.kuhn@nist.gov

Y. Lei
University of Texas at Arlington, Arlington, TX 76019, USA
e-mail: ylei@cse.uta.edu

D. E. Simos (✉)
SBA-Research, 1040 Vienna, Austria
e-mail: dsimos@sba-research.org

Published online: 12 April 2021

Birkhäuser

1 Introduction

Statistical design of experiments (DoE) were initially developed to improve agricultural production, an urgent concern in some parts of the world of the early twentieth century. DoE methods were adapted and optimized for experiments with animals, for manufacturing processes, for drugs development, and for biomedical research. Factorial experiments are a subclass of DoE popular in chemical and hardware manufacturing industries. In the twenty-first century software has become ubiquitous in tools, systems, and methods for science, engineering, medicine, commerce, human interactions, and remote work. Most high-value products have embedded software. Therefore, the assurance of quality and security of software and software-based systems has become an important world-wide concern. The cost of error and recovery from the failure of a single critical system can run into many billions of dollars [142]. The ACM Forum on Risks (Website) [1] reports many examples of computer failures and their effects on the public. Combinatorial testing (CT) is a modification of factorial experiments that complements other methods for testing, verification, and validation to assure the quality and security of software and software-based systems.

In Sect. 2, we review briefly the objectives and methods of factorial experiments. An orthogonal array (OA) is a mathematical object which may be used as the design matrix of a factorial experiment [53]. Also, the design matrix of an efficient factorial experiment is an OA or includes an OA as a subset. CT began with the use of OAs to test software-based systems. Covering arrays (CAs) are generalizations of OAs that are better suited for CT. In Sects. 3 and 4 we discuss OAs and CAs. Section 5 offers a review of CT. It includes abundant reference for further details. In Sect. 6, we discuss the main differences between factorial experiments and combinatorial testing. CT has become practical because various tools to generate test suites for CT are now available. In Sect. 7, we review a free research tool for generating test suites for CT. In Sect. 8, we comment on the application of CT to security testing of software-based systems. In Sect. 9, we restate the main points and comment on three areas for further research and application of CT.

2 Factorial Experiments (FEs)

A factorial experiment (FE) is a statistically designed investigation of a natural or man-made system (that is sufficiently-stable) in which the system is run (executed) for some input variables, called factors, each having plural test values (levels or settings), and the corresponding output variables, called responses, are measured. The generic object in a factorial experiment is to form an empirical statistical model for the relationship between the values of input factors and the output response. A sequence of factorial experiments is often required to form a statistical model. An experimental run (test run or test case or treatment) is a combination of one test value for each input factor. A batch of experimental runs is called the design of experiment or design matrix (or test suite). The chronological order of running the test cases (or allocation of treatments to the experimental units) is randomized to avoid the results of experiment from being contaminated by systematic (non-random) effects. A test case may be run two or more times (replicated) to estimate the variance of response from experimental replication error. Table 1 displays a factorial design matrix of 18 experimental runs for 5 factors (denoted by A, B, C, D, and E) each with three possible test values (denoted by -1, 0, and 1). These test values may be codes for 'low', 'center' and 'high' levels of continuous-valued factors. The text books on the design and data analysis of factorial experiments include the following classics: Fisher [35], Kempthorne [55], Cochran and Cox [19], Searle [103], Graybill [43], Box et al. [12], Snedecor and Cochran [117], Box and Draper [13], Montgomery [83], Hinkelmann and Kempthorne [49], Wu and Hamada [132], and some other more recent books, including later editions of some classics. A very brief reminder on factorial experiments is given below. More thorough description can be found in the references cited above.

Factorial design of experiments includes full factorial, fractional factorial, and response surface modeling designs. If k input factors have v test values (levels) each then there are v^k possible test cases. A full factorial design matrix includes all v^k test cases. Full factorial designs where $v = 2$ are most popular. Experiments where $v = 3$ or

Table 1 Design matrix of a factorial experiment for five factors each with three possible values

	A	B	C	D	E
1	− 1	− 1	− 1	− 1	1
2	1	− 1	− 1	− 1	− 1
3	− 1	1	− 1	− 1	− 1
4	1	1	− 1	− 1	1
5	− 1	− 1	1	− 1	− 1
6	1	− 1	1	− 1	1
7	− 1	1	1	− 1	1
8	1	1	1	− 1	− 1
9	− 1	− 1	− 1	1	− 1
10	1	− 1	− 1	1	1
11	− 1	1	− 1	1	1
12	1	1	− 1	1	− 1
13	− 1	− 1	1	1	1
14	1	− 1	1	1	− 1
15	− 1	1	1	1	− 1
16	1	1	1	1	1
17	0	0	0	0	0
18	0	0	0	0	0

where the factors have unequal numbers of test values (that is, mixed-level full factorial designs) are also used. In a two-level ($v = 2$) full factorial experiment, the effect of a factor is the change in response produced by a change in the test value of the factor while the values of the other factors are fixed. The main effect of a factor A is the average effect of that factor for all values of each other factor in the experiment. A two-factor (linear-by-linear) interaction effect A × B is the difference in the conditional main effects of factor A corresponding to the two values of the factor B averaged for all values of each other factor. A three-factor interaction effect A × B × C is the difference in the conditional two-factor interaction effects A × B corresponding to the two values of the factor C averaged for all values of each other factor. The main effects and interaction effects are collectively referred to as factorial effects. The statistical idea of three-factor and higher order interaction effects may be difficult to interpret physically.

The number of test cases 2^k increases exponentially with the number of factors k. The cost of conducting an experiment usually increases with the number of test cases to be run. Therefore, experimenters are concerned with minimizing the number of test cases without greatly compromising the information obtained about important factorial effects. A 2^{k-p} fractional factorial design, where p is less than k, is a $1/2^p$ fraction of the 2^k test cases of a full factorial design. The 2^{k-p} test cases have the same values for p independent "defining relations". In Table 1, the experimental runs 1 to 16 form a $2^{5\text{-}1}$ fractional-factorial design. For this half-fraction of all possible $2^5 = 32$ test cases, the "defining relation" is that the product of the values (− 1 or 1) of the five factors is always 1. (In the complementary 16 of the 32 test cases, the product of the five values is always − 1.) This experiment plan enables estimation of the five main effects and the ten two-factor interaction effects under the assumption that all three-factor and higher order interaction effects can be neglected. In Table 1, two center points (0, 0, 0, 0, 0) are appended to check for the curvature of the output response and to assess variation from experimental replication error.

Response surface modeling (RSM) designs, such as central composite designs and Box-Behnken designs, are also an important category of factorial designs of experiments. In a central composite design, center points and axial points are appended to a full or fractional factorial design matrix. The corresponding statistical model includes quadratic effects in addition to the main effects and (linear-by-linear) interaction effects. The design of a factorial

Table 2 Latin square of order 3

	i	ii	iii
1	A	B	C
2	B	C	A
3	C	A	B

Table 3 Another Latin square that is orthogonal to the Latin square of Table 2

	i	ii	iii
1	a	b	c
2	c	a	b
3	b	c	a

experiment is always written in a matrix form as shown in Table 1, where the columns represent the input factors, the entries in the columns represent the test values and the rows represent the experimental runs (test cases).

A statistically designed experiment is paired with a hypothesized statistical model, which is the basis for interpreting the data. Usually, the statistical model is a special case of the general linear model $\boldsymbol{y} = \mathbf{X}\beta + \boldsymbol{e}$ for each response vector $\boldsymbol{y}$, where $\mathbf{X}$ is a known model matrix, β is a vector of unobservable parameters, and $\boldsymbol{e}$ is an unobservable random vector, representing the experimental error. The error vector $\boldsymbol{e}$ is usually assumed to have a multivariate normal distribution with expected value $\mathrm{E}(\boldsymbol{e}) = 0$ and variance $\mathrm{V}(\boldsymbol{e}) = \sigma^2\mathbf{V}$ where σ^2 is an unknown parameter and $\mathbf{V}$ is a known positive definite matrix. The matrix $\mathbf{V}$ is often taken be the identity matrix $\mathbf{I}$. The number of elements in the parameter vector β is at least one more (for the overall average response) than the number of input factors (that is, columns of the design matrix). For the design matrix of 5 columns in Table 1, the model matrix $\mathbf{X}$ may have 16 columns, one for the overall average, five for the main effects, and ten for the two-factor interaction effects. The elements of the model matrix $\mathbf{X}$ may be 1's and 0's representing presence or absence of the elements of parameter vector β; coded levels (such as -1, 0, and 1); or actual values of the input factors; or other known values. The general linear model includes both analysis-of-variance (ANOVA) models and regression models. Challenges in statistical modeling are discussed in books such as Miller [82] and Freedman [37]. Generalizations of the linear model $\boldsymbol{y} = \mathbf{X}\beta + \boldsymbol{e}$ are discussed in McCulloch and Searle [79].

After running (executing) the system for each test case of the design matrix, the measured value for each output response is determined. The model matrix and the measured data on a response variable are used to estimate the unknown parameters (that is, the elements of β and σ^2). The estimates of the elements of β include factorial effects. The input factors and the output responses may be transformed to a scale for which the general linear model is a better fit to the realized data. The hypothesized statistical model and the estimated values of the unknown parameters form an empirical statistical model for the relationship between the values of input factors and the output response.

Generally, the empirical statistical model is used for one or more of the following four practical ends. (i) Determine which test values of input factors give a more favorable output response (this is called comparison). (ii) Determine which few (often, 2 to 4) of the k input factors (where k is more than 4) have the greatest effects on the output response (this is called screening). Then, further investigation is focused on such important factors. (iii) Determine a parsimonious empirical statistical model that is sufficiently good to predict the output response of the system for specified values of the input factors in some neighborhood of the test values that were used in the experiment (this is called prediction). (iv) Determine optimum values of the input factors for which the output responses of the system are favorable (this is called optimization). Thus, factorial experiments are used for comparison, screening, prediction, and optimization with respect to a target system of interest.

Table 4 Graeco-Latin square of order 3

	i	ii	iii
1	Aa	Bb	Cc
2	Bc	Ca	Ab
3	Cb	Ac	Ba

Table 5 Orthogonal array constructed from two mutually orthogonal Latin squares

	R	C	U	L
1	1	i	A	a
2	1	ii	B	b
3	1	iii	C	c
4	2	i	B	c
5	2	ii	C	a
6	2	iii	A	b
7	3	i	C	b
8	3	ii	A	c
9	3	iii	B	a

3 Orthogonal Arrays (OAs)

A Latin square of order v is an arrangement of v symbols in v^2 cells arranged in v rows and v columns, such that every symbol appears in each row and each column. Table 2 is a Latin square of three upper-case letters used as symbols (A, B, C) and Table 3 is another Latin square of three lower-case symbols (a, b, c). Two Latin squares of the same order v are said to be mutually orthogonal to each other if when superimposed on one another, every pair of symbols (consisting of one symbol from each Latin square) appears exactly once. A superimposition of two mutually orthogonal Latin-squares is referred to as a Graeco-Latin square. In Table 4, the two Latin squares of upper-case symbols (A, B, C) and lower-case symbols (a, b, c) are superimposed. In Table 4 every pair of upper-case symbols (A, B, C) and lower-case symbols (a, b, c) appears exactly once. Thus, Tables 2 and 3 are mutually orthogonal Latin squares, and Table 4 is a Graeco-Latin square. The 9 cells of Table 4 show a balanced arrangement of 4 variables: row numbers (R) with values (1, 2, 3), column numbers (C) with values (i, ii, iii), upper-case symbols (U) with values (A, B, C), and lower-case symbols (L) with values (a, b, c). Every value of each variable appears with each value of the other three variables exactly once. Table 5 displays the Table 4 (Graeco-Latin square of order 3) as a matrix of 9 rows and 4 columns. In Table 5, every possible pair of values appears exactly once. Table 6 displays the Table 5 in a canonical form using the symbols (0, 1, 2) for the values of each variable. Table 6 is said to be an orthogonal array (OA) of 9 rows, 4 columns, 3 symbols in each column, and strength 2, denoted as OA(9, 4, 3, 2). The strength 2 means that in every one of the six pair of columns ((1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)), each of the $3^2 = 9$ possible pairs of symbols ((0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)) appears with the same frequency. The same frequency is called index. The index of OA(9, 4, 3, 2) is one.

An orthogonal array $\mathrm{OA}(N, k, v, t)$ is an $N \times k$ matrix of entries from a set of v symbols (elements) $V = (0, 1, \ldots, (v-1))$ and strength t if every $N \times t$ sub-matrix contains each possible t-tuple of symbols the same number of times, called index denoted by λ. Since the index λ is determined by the other parameters N, k, v, and t, it is not mentioned ($\lambda = N/v^t$). An orthogonal array of strength t is necessarily an OA of strength less than t. A permutation of the rows, or the columns, or the symbols $(0, 1, \ldots, (v-1))$ of any column of an orthogonal array (OA) results in an OA with the same parameters. In place of the list notation $\mathrm{OA}(N, k, v, t)$, the exponential notation $\mathrm{OA}(N, v^k, t)$ is also used.

Table 6 Orthogonal array OA($N = 9, k = 4, v = 3, t = 2$) in canonical form

	1	2	3	4
1	0	0	0	0
2	0	1	1	1
3	0	2	2	2
4	1	0	1	2
5	1	1	2	0
6	1	2	0	1
7	2	0	2	1
8	2	1	0	2
9	2	2	1	0

An OA is a mathematical object that satisfies the combinatorial properties stated above. However, it may be used as a design matrix of a fractional-factorial experiment of k factors with v values each in N test runs. For example, a full factorial experiment of $k = 4$ factors each with $v = 3$ values requires $v^k = 3^4 = 81$ test cases. The OA$(9, 4, 3, 2) \equiv$ OA$(9, 3^4, 2)$ displayed in Table 6 may be regarded as a fraction $N/v^k = 9/81 = 1/9$ of the design matrix of a 3^4 full factorial experiment. This highly fractionated plan may be useful when in the associated statistical model, the effects of all 4 factors are additive; that is, all interaction effects are neglected.

An OA(N, k, v, t) is a fixed-level orthogonal array, since every column has the same fixed number v of distinct symbols (elements). When each column has two distinct symbols, the OA is called a binary OA. The concept of a fixed-level OA can be extended to a mixed-level OA where not-all-columns have the same number of distinct elements. Suppose the k factors have $v_1, \ldots, v_k$ elements, respectively, then a mixed-level OA may be denoted as OA$(N,k,(v_1, \ldots, v_k), t)$. When not all k values $(v_1, \ldots, v_k)$ are distinct, then in place of the list $(v_1, \ldots, v_k)$ an exponential notation may be used. An interesting example is OA$(36, 2^{11} \times 3^{12}, 2)$. This OA may be regarded as a fraction $36/(2^{11} \times 3^{12}) = 1/30,233,088$ of the design matrix of a full factorial experiment of 23 factors where 11 factors have 2 distinct values each and 12 factors have 3 distinct values each. The OA$(36, 2^{11} \times 3^{12}, 2)$ appears in Taguchi [119]. It was first constructed by Masuyama [78] from a "difference set" which Seiden [106] found (by trial and error) and used to construct OA$(36, 3^{13}, 2)$. In 1985, an OA$(36, 2^{11} \times 3^{12}, 2)$ was used to investigate the main effects of 17 factors in an experiment on wave soldering process [73].

The concept of orthogonal arrays was defined by Rao [100]. However, Taguchi [119] promulgated the use of OAs for industrial experiments [52]. A library of known OAs is offered by Sloane (Website) [114]. In the Sloane's library, most OAs have strength $t = 2$; OAs of strength $t = 3$ are binary, and very few OAs have strength greater than $t = 3$. Additional information about OAs and their relationship with design matrices of factorial experiments can be found in Raghavarao [98], Raktoe et al. [99], Hedayat et al. [48], and the references cited there.

4 Covering Arrays (CAs)

Sloane [115] generalized the concept of an orthogonal array of index one to a mathematical object called covering array (CA). A covering array CA$(N; t, k, v)$ is an $N \times k$ matrix of entries from a set of v symbols (elements) $V = (0, 1, \ldots, (v - 1))$ and strength t if every $N \times t$ sub-matrix contains each possible t-tuple of symbols at least once. With k columns there are k-choose-t, C(k, t), sets of t columns each. In every set, the number of possible t-tuples of symbols is v^t. The t-tuples of symbols for one set of t columns are distinct from the t-tuples of symbols for another set of t columns, although they may superficially appear alike. So, the total number of distinct t-tuples of symbols is C$(k, t) \times v^t$. Each distinct t-tuple must appear at least once in the N rows for the $N \times k$ matrix to be a CA. A permutation of the rows, or the columns, or the symbols $(0, 1, \ldots, (v - 1))$ of any column of a covering

Table 7 CA($N = 24$; $t = 4$, $k = 12$, $v = 2$) generated by the IFS-greedy algorithm [54]

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	1	0	0	1	0	1	1	0	1	0
2	1	1	0	0	1	0	1	1	0	0	1	0
3	1	0	0	0	1	1	1	0	1	1	0	0
4	0	1	1	0	0	0	1	0	1	1	1	0
5	0	1	1	1	0	0	0	1	0	0	1	1
6	1	1	1	1	0	1	1	0	0	0	0	0
7	1	1	0	0	0	1	0	0	0	1	1	1
8	1	0	1	1	1	0	0	0	0	1	1	0
9	1	1	0	1	0	0	0	1	1	1	0	0
10	1	0	1	0	0	0	1	1	0	1	0	1
11	1	0	0	1	1	1	0	1	0	0	0	1
12	1	1	1	0	1	0	0	0	1	0	0	1
13	0	0	1	0	1	1	1	0	0	0	1	1
14	0	0	1	1	0	1	0	0	1	1	0	1
15	0	1	0	0	0	1	1	1	1	0	0	1
16	1	1	1	1	1	1	1	1	1	1	1	1
17	0	0	0	0	1	0	0	1	1	1	1	1
18	0	0	1	1	1	0	1	1	1	0	0	0
19	0	1	1	0	1	1	0	1	0	1	0	0
20	1	0	0	1	0	0	1	0	1	0	1	1
21	0	1	0	1	1	0	1	0	0	1	0	1
22	0	1	0	1	1	1	0	0	1	0	1	0
23	0	0	0	1	0	1	1	1	0	1	1	0
24	0	0	0	0	0	0	0	0	0	0	0	0

array results in a CA with the same parameters. In place of the list notation CA(N; t, k, v), the exponential notation CA(N; t, v^k) is also used. In the notation CA(N; t, k, v) for a covering array, the strength t is listed after the number of rows, whereas in the notation OA(N, k, v, t) for an orthogonal array the strength t is listed last. The concept of a fixed-level covering array CA(N; t, k, v) can be extended to a mixed-level covering array where not all columns have the same number of distinct elements. Suppose the k factors have $v_1, \ldots, v_k$ elements each then a mixed-level covering array may be denoted as CA(N; t, k, ($v_1, \ldots, v_k$)). When not all k values ($v_1, \ldots, v_k$) are distinct, an exponential notation may be used in place of the list notation.

Table 7 displays CA(24; 4, 12, 2), a covering array of $N = 24$ rows, strength $t = 4$, $k = 12$ columns, and $v = 2$ symbols in each column. This example of CA(24; 4, 12, 2) was generated by an algorithm described in Kampel and Simos [54]. An earlier reference for the construction of a CA(24; 4, 12, 2) is Yan and Zhang [135]. In each column of Table 7, twelve elements are 0 and twelve are 1. Table 8 displays the 12 rows of the first 11 columns of Table 7 corresponding to the element 0 in the column number 12. Interestingly, Table 8 is a covering array CA(12; 3, 11, 2) of strength $t = 3$. The elements of any single column of the 12 columns of CA(24; 4, 12, 2) can be used to form two covering arrays of strength 3 like Table 8.

The minimum number N of rows required to produce a CA(N; t, k, v) is called covering array number denoted as CAN(t, k, v). A covering array number CAN(t, k, v) may be determined (known) but the corresponding covering array CA(N; t, k, v) may be difficult to produce. When a CA(N; t, k, v) has been produced, one knows an upper

Table 8 CA($N = 12$; $t = 3$, $k = 11$, $v = 2$) derived as rows corresponding to the element 0 in the column 12 of Table 7

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	0	0	1	0	1	1	0	1
2	1	1	0	0	1	0	1	1	0	0	1
3	1	0	0	0	1	1	1	0	1	1	0
4	0	1	1	0	0	0	1	0	1	1	1
5	1	1	1	1	0	1	1	0	0	0	0
6	1	0	1	1	1	0	0	0	0	1	1
7	1	1	0	1	0	0	0	1	1	1	0
8	0	0	1	1	1	0	1	1	1	0	0
9	0	1	1	0	1	1	0	1	0	1	0
10	0	1	0	1	1	1	0	0	1	0	1
11	0	0	0	1	0	1	1	1	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0

bound for CAN(t, k, v). Colbourn (Website) [24] has posted the best known upper bounds on CAN(t, k, v) for $t = 2, 3, 4, 5$, and 6; k up to 20,000 for $t = 2$, and k up to 10,000 for $t = 3, \ldots, 6$; and v up to 25.

The NIST covering array tables (Website) [87] display CA(N; t, k, v) for $t = 2, 3, 4, 5$, and 6 and $v = 2, 3, 4, 5$, and 6 (except for $t = v = 6$). These tables were developed by Michael Forbes during the summer of 2007 [36]. Torres-Jimenez (Website) [124] displays a collection of binary ($v = 2$) covering arrays CA(N; t, k, v) for $t = 2, 3, 4, 5$, and 6. A survey of binary covering arrays appears in Lawrence et al. [64]. Additional information about CAs can be found in Colbourn [25], Hartman and Raskin [47], Colbourn and Dinitz [26], Torres-Jimenez and Izquierdo-Marquez [125], and Panario et al. [92] and the references cited there.

Useful orthogonal arrays of sufficiently small number of rows N exist only for certain conforming values of the parameters k, $(v_1, \ldots, v_k)$ and t. However, useful covering arrays can be constructed for any values of k, $(v_1, \ldots, v_k)$, and t. It is difficult to construct useful orthogonal arrays OA(N, k, $(v_1, \ldots, v_k)$, t) of strength t greater than two or three. However, covering arrays CA(N; t, k, $(v_1, \ldots, v_k)$), for any practical value of the strength t can be constructed. (In combinatorial testing, CAs of strength t up to 6 may be required.) An OA(N, k, v, t) of index $\lambda = 1$ is the smallest possible covering array. However, OAs of index $\lambda = 1$ exist only for exceptional values of the parameters N, k, v, and t [15]. Except when an OA of index $\lambda = 1$ exists, for a given number of factors k with values $(v_1, \ldots, v_k)$ and strength t, a CA of fewer number of rows N than the number of rows in a corresponding OA can usually be constructed. Orthogonal arrays OA(N, k, $(v_1, \ldots, v_k)$, t) are constructed using combinatorial design theory (finite plane projective and Euclidean geometries). To construct a covering array CA(N; t, k, $(v_1, \ldots, v_k)$), both computational algorithms and combinatorial design theory are used, alone or in combination. CAs constructed from combinatorial design theory usually have fewer rows than the corresponding CAs generated by computational algorithms. In the following subsections, we describe three special types of covering arrays.

4.1 Variable-Strength Covering Array

A covering array of strength t is also a CA of strength less than t. A CA(N; t, k, v) includes some combinations of more than t values; for example, it includes the number N of the v^k possible k-tuples (where k is greater than t). It is possible to construct a CA of two or more strengths. For example, a CA(N; t, k, v) can be extended to obtain a variable-strength covering array of $\tilde{N}$ rows where the first k_1 columns 1,…, k_1 have strength t_1, the next k_2 columns $k_1 + 1, \ldots, k_1 + k_2$ have strength t_2, and $k = k_1 + k_2$. Such a variable-strength covering array may be denoted as CA($\tilde{N}$; $(t_1; 1, \ldots, k_1)$, $(t_2; k_1 + 1, \ldots, k_1 + k_2)$, k, v) or simply as CA($\tilde{N}$; t = Variable, k, v). Table 9 displays

Table 9 CA($N = 17$; $t =$ Variable, $k = 11$, $v = 2$) constructed from Table 8 by appending the rows 5, 11, 12, 13 and 15 of Table 7

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	0	0	1	0	1	1	0	1
2	1	1	0	0	1	0	1	1	0	0	1
3	1	0	0	0	1	1	1	0	1	1	0
4	0	1	1	0	0	0	1	0	1	1	1
5	1	1	1	1	0	1	1	0	0	0	0
6	1	0	1	1	1	0	0	0	0	1	1
7	1	1	0	1	0	0	0	1	1	1	0
8	0	0	1	1	1	0	1	1	1	0	0
9	0	1	1	0	1	1	0	1	0	1	0
10	0	1	0	1	1	1	0	0	1	0	1
11	0	0	0	1	0	1	1	1	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0
13	0	1	1	1	0	0	0	1	0	0	1
14	1	0	0	1	1	1	0	1	0	0	0
15	1	1	1	0	1	0	0	0	1	0	0
16	0	0	1	0	1	1	1	0	0	0	1
17	0	1	0	0	0	1	1	1	1	0	0

a variable-strength covering array CA($N = 17$; $t =$ Variable, $k = 11$, $v = 2$) whose first four columns 1, 2, 3, 4 have strength 4 and the next seven columns 5, 6, 7, 8, 9, 10, 11 have strength 3. In the first four columns each of the $2^4 = 16$ quadruples of symbols ((0, 0, 0, 0), (0, 0, 0, 1), . . . , (1, 1, 1, 0), (1, 1, 1, 1)) appears at least once; the quadruple (0, 1, 1, 0) appears twice (in rows 4 and 9) and the other appear exactly once. The first twelve rows of Table 9 are identical to Table 8. Table 9 was constructed by appending to Table 8 rows 5, 11, 12, 13 and 15 of CA(24; 4, 12, 2) shown in Table 7. Variable-strength covering arrays are useful in combinatorial testing.

4.2 Constrained Covering Array

Certain combinations of the elements (symbols representing test values of input factors) may be invalid (physically or logically). A row (representing a test case) that includes an invalid combination is invalid. An invalid combination destroys an orthogonal array. However, a constrained covering array that avoids invalid combinations can be constructed. For example, consider the covering array CA(12; 3, 11, 2) shown in Table 8. It contains each of the $C(11, 3) \times 2^3 = 1320$ distinct triples of symbols at least once. (A triple of symbols, for example (0, 0, 0), in columns 1, 2, and 3 is distinct from the triple (0, 0, 0) in columns 1, 2, and 4.) Suppose that for some reason, in the three pairs of columns (5, 6), (7, 8) and (9, 10) the pair of elements (1, 1) is invalid. Then the eight rows (2, 3, 4, 7, 8, 9, 10, and 11) of the twelve rows in Table 8 are invalid. Only four rows (1, 5, 6, and 12) remain valid. The number of distinct triples involving the three invalid pairs of elements (1, 1) is $3 \times 9 \times 2 = 54$. Thus, the number of distinct valid triples is $1320 - 54 = 1266$. The four valid rows of CA(12; 3, 11, 2) include only 600 of the 1266 valid triples. Table 10 displays a constrained covering array CA(20; 3, 11, 2) of 20 rows which includes all 1266 valid triples of combinations (subject to the constraint that the pair of elements (1, 1) in the three pairs of columns (5, 6), (7, 8) and (9, 10) are forbidden). The four valid rows from Table 8 are preserved. This constrained covering array was generated by extending the valid rows using a tool discussed in Sect. 7. Whether all 1266 valid triples can be covered with less than 20 rows remains a challenge. Constrained covering arrays are useful in combinatorial testing.

Table 10 Constrained CA($N = 20$; $t = 3$, $k = 11$, $v = 2$) which avoids the pair of elements (1, 1) in the three pairs of columns (5, 6), (7, 8) and (9, 10). The four valid rows from Table 8 are preserved

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	0	0	1	0	1	1	0	1
2	1	1	1	1	0	1	1	0	0	0	0
3	1	0	1	1	1	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	1	1	0	1	0	1	0	0
6	0	1	0	1	1	0	0	1	1	0	1
7	1	1	0	0	1	0	1	0	0	1	1
8	0	0	1	1	0	1	1	0	1	0	1
9	0	1	1	0	1	0	1	0	1	0	0
10	1	1	0	1	0	1	0	1	0	1	0
11	0	1	0	0	0	1	1	0	1	0	1
12	0	0	1	0	1	0	0	1	0	1	0
13	1	1	1	1	0	0	0	1	1	0	0
14	0	0	0	0	0	1	1	0	0	1	1
15	0	1	1	1	0	0	1	0	0	1	0
16	0	1	0	0	0	1	0	1	0	1	1
17	0	1	1	0	0	1	0	0	1	0	0
18	1	0	0	1	1	0	0	1	0	0	1
19	1	0	1	0	0	1	0	0	0	1	0
20	1	1	1	1	0	0	1	0	1	0	1

4.3 Some Binary OAs are CAs of Higher Strength and Some CAs are OAs of Lower Strength

Some two-level ($v = 2$) fractional factorial experiments are based on design matrices attributed to Plackett and Burman [97]. Plackett-Burman (P-B) designs are binary orthogonal arrays OA(N, k, $v = 2$, $t = 2$) of strength two, where the number of rows N is a multiple of 4 and the number of columns $k = N - 1$. They are a subclass of OAs obtained from Hadamard matrices [48]. It is known that some P-B designs for $N = 12, 16, 20, 24, 28, 32,$ and 36 have the property that every set of three columns contains at least one 2^3 full factorial design [72]. Thus, these orthogonal arrays of strength $t = 2$ are covering arrays CA(N; $t = 3$, k, $v = 2$) of higher strength $t = 3$. Sloane's library of OAs includes binary orthogonal arrays OA(40, 20, 2, $t = 3$) and OA(56, 28, 2, $t = 3$) of strength $t = 3$. It can be checked that these OAs of strength $t = 3$ are covering arrays CA(40, $t = 4$, 20, 2) and CA(56, $t = 4$, 28, 2) of higher strength $t = 4$. Thus, some binary orthogonal array OA(N, k, v, $t-1$) of strength $t - 1$ are covering arrays CA(N; t, k, v) of strength t.

The binary covering array CA(24; $t = 4$, 12, 2) of strength $t = 4$ displayed in Table 7 turns out to be an orthogonal array OA (24, 12, 2, $t = 3$) of lower strength $t = 3$. The binary covering array CA(12; $t = 3$, 11, 2) of strength $t = 3$ displayed in Table 8 is an orthogonal array OA (12, 11, 2, $t = 3$) of lower strength $t = 2$. Thus, some binary covering arrays CA(N; t, k, v) of strength t are orthogonal array OA(N, k, v, $t - 1$) of strength $t - 1$. This observation is possible because explicit tables of binary covering arrays are available.

In this paper, we discuss the use of covering arrays for testing software-based systems. In general, covering arrays are useful whenever a large input space or experimental space is to be explored with a small number of test cases to detect combinations that are important in some sense (see, for example, [67,107]).

5 Combinatorial Testing (CT)

Combinatorial testing (CT) is a methodology for testing a software-based system in which the system under test (SUT) is executed (run) for a collection of test cases. A test case consists of one test value for each of the test factors. The collection of test cases is called a test suite. In CT, the test suite satisfies the property that for every subset of t out of all k test factors (where $k \geqslant t$) each possible combination of the t-test values (t-tuples) is present in at least one test case [59]. The number t is the strength of the test suite (where $t = 2, 3, 4, \ldots$). Thus, CT is based on covering arrays. A test suite for CT is generated by a computational tool or constructed from an available covering array (by assigning the factors to the columns and associating the test values with the entries in the columns). To a test suite determined from a CA, a tester may append additional test cases based on technical judgment. For each test case that is executed, the behavior of the system is monitored. The expected (correct) behavior of the system is pre-determined. The observed (actual) behavior is compared against the expected. A complete test case consists of both the description of the test value for each test factor and the corresponding expected system behavior. When the actual behavior agrees with the expected, the system has passed the test case. When the actual behavior disagrees with the expected, the system has failed the test case. In most applications of CT, the result for a test case (pass or fail) does not change when that test case is repeated (that is, run again). Certain combinations of the test values may be logically or physically forbidden. Such forbidden combinations are excluded from the test suite. When the system fails for one or more test cases, efforts are made to locate, from the pass/fail data, one or more combinations of the values of a least number of factors which may have induced the failures. An n-way combinatorial interaction fault is a combination of n test values (where $n = 1, 2, 3, \ldots$) whose presence in a test case induces a failure of the SUT (that is, produces a behavior that is different from the pre-determined expected behavior). The term 'n-way combinatorial interaction fault' is generally abbreviated to 'n-way interaction fault'. The purpose of combinatorial t-way testing is to detect n-way interaction faults for n less than or equal to a chosen strength t. We will discuss the choice of strength t in combinatorial testing. (An n-way interaction fault for $n > t$ may also be detected if it is present in the test suite.) The effort to isolate one or more n-way interaction faults from the pass/fail data is called combinatorial interaction fault localization. We will abbreviate the term 'combinatorial interaction fault localization' to 'combinatorial fault localization'. Combinatorial fault localization may require defining and running additional test cases. Once a failure-inducing interaction fault has been detected, the underlying defect in the source code of the SUT is searched and corrected. Thus, combinatorial testing is a methodology for detecting interaction faults in a software-based system. (In CT, an n-way interaction fault is a combination of n test values. There is no connection between an interaction fault and the statistical concept of interaction effects in a factorial experiment.)

As an example, suppose the object of investigation is a software which receives inputs from a user and displays the corresponding output at the monitor of a computer. A user inputs the values of 10 factors A, B, C, D, E, F, G, H, I, and J and the software outputs the corresponding result. Suppose each of the ten input factors can have two possible values, which are denoted by 0 and 1. A test case is a combination of one of the two values (0 or 1) for each of the ten input factors. Thus, a test case is an input vector of ten zero or one values. The correct output result for each test vector is pre-determined. For each test vector, the software must produce the pre-determined correct output result. With ten factors each having two possible values, the total number of possible test cases is $2^{10} = 1024$. Running all possible test cases is called exhaustive testing. (Exhaustive testing corresponds to running a full factorial experiment). Exhaustive testing covers all possible n-way combinations for every value of n from 1 to 10. Often, exhaustive testing is neither practical nor necessary. In the present case, suppose it is sufficient to test for n-way combinations for $n = 1, 2$, and 3. The number of single test values (trivially, 1-way combinations, $A = 0, A = 1, \ldots, J = 0, J = 1$) is $10 \times 2 = 20$. There are $C(11, 2) = 45$ pairs of factors (AB, AC, . . ., IJ). Each pair can have $2^2 = 4$ possible value (00, 01, 10, 11). So, the number of 2-way combinations (pairs of test values) is $C(10, 2) \times 2^2 = 180$. The number of 3-way combinations (triples of test values) is $C(10, 3) \times 2^3 = 960$. Triples contain pairs which contain single test values. Suppose we assign the ten input factors A, B, C, D, E, F, G, H, I, and J to the first 10 of the 11 columns of the strength $t = 3$ covering array CA(12; 3, 11, 2) displayed in Table 8, associate the corresponding test values 0 and 1 with the entries in the columns, and delete the eleventh column. Then the 12 rows of CA(12; 3, 11, 2) define 12 of the 1024 test cases which cover all 960 triples of the test values.

Table 11 Results from the study of [71]

Package	Method	Number of test cases	Number of bugs found	Did CT find FT bugs?
DP-I	FT	98	2	
	CT	49	6	Yes
DP-II	FT	102	1	
	CT	77	5	Yes
DP-III	FT	116	2	
	CT	80	7	Missed 1
DP-IV	FT	122	2	
	CT	90	4	Yes

Thus, by running a test suite of 12 test cases determined by using CA(12; 3, 11, 2) it is possible to detect any n-way interaction fault for n up to the strength $t = 3$. For detecting any n-way interaction fault for n greater than the strength $t = 3$, more than 12 test cases would typically be required.

The covering array CA(12; 3, 11, 2) contains some n-way combinations for n greater than 3. If any one of these n-way combinations turns out to be an interaction fault, then its existence may also be revealed. Suppose a software tester wants to assure that no 4-way interaction fault involving the first four input factors A, B, C, and D exists. Table 9 is a variable-strength covering array CA(17; (4; 1, . . . , 4), (3; 5, . . . , 11), 11, 2) of two strengths where the first 4 columns 1, 2, 3, 4 have strength 4 and the next 7 columns 5, 6, 7, 8, 9, 10, 11 have strength 3. By using a test suite of 17 test cases constructed from CA(17; (4; 1, . . . , 4), (3; 5, . . . , 11), 11, 2), it is possible to also detect any 4-way interaction fault involving the four input factors A, B, C, and D. Testing for 4-way interaction faults involving all ten input factors would require 24 test cases (because the covering array number CAN for $t = 4$, $k = 10$ and $v = 2$ is 24, [24].

Now, suppose the object of investigation is a text processing software. Suppose the ten input factors are font options: A: Bold, B: Italic, C: Underline, D: Strikethrough, E: Subscript, F: Superscript, G: Outline-effect, H: Shadow-effect, I: Large, and J: Small. Suppose each of the font options has two possible values: off (denoted by 0) and on (denoted by 1). The following three pairs of font options are mutually exclusive: (i) Subscript and Superscript (E and F), (ii) Outline-effect and Shadow-effect (G and H), and (iii) Large and Small (I and J). Thus, the pair of test values (1, 1) is forbidden for the three pairs of input factors (E and F), (G and H), and (I and J). The CA(20; 3, 11, 2) displayed in Table 10 is a constrained covering array which avoids the pair of elements (1, 1) in the three pairs of columns (5, 6), (7, 8) and (9, 10). By assigning the ten input factors to the first ten of the eleven columns of this constrained covering array a test suite of 20 valid test cases can be constructed. With this test suite it is possible to detect any n-way interaction fault for n up to 3, while avoiding the forbidden pairs.

The NIST Combinatorial testing page (Website) [88] lists over forty application papers. Most of these papers describe application of CT for detecting errors (bugs) in a software-based system. For example, Li et al. [71] describe an application in which four software packages (DP-I, DP-II, DP-III, and DP-IV) which control the dashboard of a subway control system were tested. A software testing company had used a function-coverage-based technique (FT) to test the four packages and had found some bugs. The company allowed a subset of the authors (Li, Gao, Wong, Yang, and Li) to re-test the same four packages using combinatorial testing (CT). The original testing took two months. Combinatorial testing took two weeks. The results from Li et al. [71] are reproduced in Table 11. CT detected more bugs with fewer test cases, but failed to detect one bug previously found by FT.

Combinatorial testing is based on covering arrays (CAs) rather than orthogonal (OAs) for the following reasons. (i) CAs can be constructed for any number of factors k, any unequal numbers of test values $(v_1, \ldots, v_k)$, and any strength t. (ii) For a given number of factors k with values $(v_1, \ldots, v_k)$ and strength t, a CA of fewer number of rows N than the number of rows in a corresponding OA can usually be constructed (except when an OA of index one is available). (iii) In generating test suites based on CAs, forbidden combinations can be excluded. (iv)

Variable-strength CAs can be used to cut down the number of test cases. (v) The balancing property of OAs (that is, in every set of t out of k columns, each t-tuple of symbols appears the same number of times) is not needed in combinatorial testing.

5.1 Generation of Test Suites for CTs

Combinatorial testing began in the 1980s as pairwise (2-way) testing using orthogonal arrays (OAs) in various organizations. We are aware of the work in Fujitsu of Japan and the descendent organizations of the AT&T Bell System in the USA. The earliest papers include the following: Sato and Shimokawa [101], Shimokawa [109], Mandl [76], Tatsumi et al. [123], and Tatsumi [122], An AT&T Bell Labs researcher and his colleagues developed a tool called OATS (Orthogonal Array Testing System) to generate test suites based on OAs of strength 2. OATS was used to test an AT&T system PMX/Starmail [14]. When another AT&T Bell Labs researcher tried to use OATS to specify "client test configurations for a local area network product", he ran into the limitations of test suites based on OAs. Some test cases were invalid and could not be run because they contained invalid (forbidden) combinations. So, he developed a tool called CATS (Constrained Array Testing System) to generate test suites which excluded forbidden combinations of the test values while minimizing the number of test cases [108]. Test suites generated by CATS were predecessors of the test suites based on constrained covering arrays.

The first publicly available tool for generating test suites based on CAs for pairwise testing was AETG, Cohen et al. [20–22]. In 1998, Lei and Tai developed a tool called IPO for generating test suites for pairwise testing based on CAs which excluded forbidden combinations of test values, Lei and Tai [65], and Tai and Lei [120]. Usefulness of CAs led to a burst of interest among mathematicians and computer scientists to develop tools for generating test suites based on CAs for pairwise (and higher strength) testing. A website maintained by Czerwonka (Website) [30] lists 37 tools (beginning with OATS, CATS, AETG and IPO) for generating test suites for combinatorial testing.

5.2 Inadequacy of Pairwise Testing

A team of NIST (National Institute of Standards and Technology, USA) researchers investigated fifteen years' worth of recall data due to failures of software embedded in medical devices and failure reports for a browser, a server, and a database system. The initial motivation of these investigations was to generate insights into the kinds of software testing that could have detected the underlying flaws (defects, errors, and other unintended mistakes) in the software to prevent the failure in use. The NIST researchers counted the numbers of 'individual factors' that were involved in the flaws underlying the actual failures. In the four systems investigated by the NIST team, 29% to 68% of the flaws involved a single factor; 70% to 97% of the flaws involved one or two factors; 89% to 99% of the flaws involved one, two or three factors; 96% to 100% of the flaws involved one, two, three or four factors; 96% to 100% of the flaws involved one, two, three, four or five factors, and no flaw involved more than six factors [62,63,128]. A lesson from the NIST investigations of the failures of real-world systems is as follows. While many factors may affect the behavior of a system, only a few are involved in any given failure. Most failures involved a single factor or interaction between two factors. Progressively fewer failures involved interactions between three, four, or more factors. The maximum number of factors involved in a failure so far observed is six. This lesson is referred to as the empirical interaction rule. In the context of combinatorial testing, the empirical interaction rule implies that combinatorial (2-way) testing (commonly known as pairwise testing) is useful but it may not always provide adequate assurance. Combinatorial (t-way) testing for t greater than 2 may be required. These conclusions are supported by other investigations including the following: Finelli [34], Bell and Vouk [8], Zhang et al. [140]. This insight motivated the authors of this paper to advance the tools and techniques for combinatorial (t-way) testing for $t \geqslant 2$.

5.3 Choice of Strength t

For a given application, how should a software tester choose the strength t for combinatorial (t-way) testing? Consider the combinatorial test structure $3^3 4^4 5^2$; that is, nine factors of which three have 3 values each, four have 4 values each, and two have 5 values each. (This happens to be the configuration options for a smart phone.) Exhaustive testing would require $3^3 4^4 5^2 = 172,800$ test cases. The number of test cases required for t-way testing for $t = 2, 3, 4, 5$, and 6 are, respectively, 29, 137, 625, 2532, and 9168 (as determined by a test generation tool discussed in Sect. 7). The number of test cases required increases by an order of magnitude as the strength t increases from 2 to 3 to 5.

The choice of strength $t = 5$ or 6 greatly increases the cost of testing and combinatorial fault localization. The expected (correct) behavior of the system would need to be pre-determined for thousands of test cases, which is an arduous task. Running the SUT for thousands of times may take much effort and a long time. If one or more test cases fail, combinatorial fault localization is required. The number of test cases for which the SUT fails tends to increase with the strength t even when the faults involve only 2 or 3 factors. So, combinatorial fault localization is substantially more difficult and costlier when t is set as 5 or 6. Interaction faults for $t = 5$ or 6 occur infrequently and they arise in very special situations. So, interaction faults involving 5 or 6 factors may not show up. In most cases, the cause of failure turns out to be an interaction fault involving four or fewer factors.

The success of CT in detecting any lurking interaction faults depends critically on the choice of the test factors and their test values. A tester may wish to test the system with different test values of the current factors or altogether different factors and test values. For the same cost as required to test for 5-way or 6-way interaction faults with the current factors and test values, it may be possible to test several times with different test values or altogether different factors and test values for interaction faults involving four or fewer factors. The latter course of action may provide more thorough assurance about more likely interaction faults involving four or fewer factors. Therefore, we suggest that in the absence of additional knowledge about possible existence of the t-way interaction faults for $t > 4$, the choice of strength t should be limited to 2, 3, and 4. Li et al. [68] and Chandrasekaran et al. [16] offered the same advice based on empirical investigations in two different contexts.

5.4 Black-Box Investigation

Methods for testing software-based systems are often categorized into black-box and white-box testing. In black-box testing, a SUT is run (executed) for various input test cases (possible inputs or configuration options of the SUT) and the corresponding outputs are used to investigate the system without reference to the internal workings. In black-box testing of a software, the source code of the software is hidden. The input test cases are determined from the specifications, requirements, and prior experience with similar systems. Both allowed and disallowed but physically (and logically) possible inputs are used. Somehow, the expected (correct) behavior of the system in response to the input test cases must be pre-determined.

White-box testing refers to checking of the internal workings of a SUT and the source code of software. White-box testing includes control flow testing (see [4]), data flow testing (see [118]), and testing for complete modified condition/decision coverage (MC/DC) (see [18]). Typically, white-box testing methods analyze the source code to derive test cases such that certain characteristics, for example, all statements, all branching conditions, or all definition-use pairs, are checked. In an intermediate category called gray-box testing, the internal workings of the system are partially known; for example, the data structures and the mathematical algorithms that are implemented in the software may be known. The available partial knowledge is used to define the test cases and check the execution behavior. Both black-box and white-box testing are required in practice. White-box testing is usually done by the software developers themselves. Black-box testing is often done by independent software verifiers and validators.

A factorial experiment is a black-box investigation because the experimenter may have limited understanding of the internal structures and workings of the system. Combinatorial testing is also a black-box technique. However,

when a combinatorial interaction fault is detected, the source code of software is required to search and correct the underlying defects. When the internal structures and workings of the system are known, or the source code of software is available, such knowledge may be used to define the test factors and their values.

5.5 Defects, Errors, and Vulnerabilities

Defects in a software-based system may loosely be categorized into two types: errors and vulnerabilities. Errors are usually referred to as bugs. An error is a defect that can make a software-based system behave in an unexpected way for some user inputs or configurations or states of the system. A vulnerability is a defect in the design, implementation, or operation and management of a software-based system that an adversary can exploit to breach privacy, security, safety, or otherwise violate the security policy of the system. Some errors can lead to vulnerabilities. Most errors tend to be ordinary bugs in coding (or implementation), although errors also occur in design and configuration of a system. Vulnerabilities can arise, for example, from mistakes such as not validating external inputs, and buffer overflow errors. It turns out that about two-thirds of the vulnerabilities tend to be implementation-related [61]. Combinatorial testing is useful for both detecting errors (bugs) and exposing vulnerabilities.

5.6 Sub-Processes of Combinatorial Testing

The software-based SUT must be bounded and described completely. When a SUT has many functions, separate testing may be needed for some functions. All documentation including specifications, user manuals, possible inputs and outputs, possible configuration options, and permitted environmental use conditions must be collected. Such documents are used to define the test cases and determine the corresponding expected behaviors of the system. When the SUT is like one or more previously tested systems, prior experience may be useful in developing a test suite for the current system. Combinatorial testing requires six sub-processes: (i) input parameter modeling, (ii) test generation, (iii) test oracle, (iv) infrastructure for executing the test cases, (v) combinatorial fault localization, and (vi) correction of the SUT.

(i) Input parameter modeling

In statistical inference, the term parameter is used for a fixed constant of a probability distribution or a statistical model which is to be estimated. In the computer science literature on software testing the input test factors for the SUT are referred to as parameters. The test settings (or levels) of the test factors are referred to as values of the parameters. Thus, input parameter modeling is the process of specifying the parameters (test factors) for testing a SUT, their discrete test values, and constraints among those test values. An input parameter model (IPM) consists of the specified parameters, test values, and constraints. An IPM defines the scope of combinatorial testing. The types of parameters used in CT include possible inputs to the SUT and configuration options of the SUT (including controllable state variables). A parameter may have a range of infinitely many possible values on a continuous scale or a wide range of discrete values. For each specified parameter, discrete test values are selected. Constraints are the restrictions that must be satisfied for a test case to be valid. For example, if one wants to check whether a web-application can work correctly with different browsers and operating systems, Internet Explorer cannot be combined with MacOS, since Internet Explorer is not supported by MacOS. The constraints identified in the IPM usually imply avoidance of certain combinations of the test values.

The effectiveness of CT depends to a large extent on the quality of IPM. Suppose a system fails only when two parameters take on a specific pair of values. If that pair of values is not captured in the IPM, CT would not be able to detect the underlying interaction fault and the defect. The following references provide more details on input parameter modeling: Ostrand and Balcer [91], Grochtmann and Grimm [46], Lott et al. [74], Grindal and Offutt [44] and Grindal et al. [45].

(ii) Test generation

Test generation refers to the process of building a test suite of a small number of test cases based on a covering array which includes all t-way combinations of the test values while avoiding the forbidden combinations (which

are dictated by the constraints stated in the IPM). The desired strength t of the test suite is chosen before test generation. Combinatorial testing has become practical because various tools for generating test suites for CT are now available, Czerwonka (Website) [30]. The following references offer more details on generation of test suites for CT: Cohen et al. [22,23], Lei et al. [66], Nie and Leung [85], Khalsa and Labiche [56], Wu et al. [133], Wagner et al. [127], and Mercan et al. [81].

(iii) Test oracle

A test oracle is a mechanism for determining whether the SUT has passed or failed the test cases. It has two parts: oracle information that represents the expected (correct) behavior of the SUT for each test case and an oracle procedure that compares the oracle information with the actual behavior. For each test case, the oracle information must be pre-determined. An execution monitor is used to assess the actual behavior of the SUT after running (executing) the test cases. The test oracle is then used to compare the actual with the expected behavior and declare whether the SUT has passed or failed the test case. Development of a test oracle is a challenging problem in all software testing not just CT. The following references discuss the test oracle problem: Peters and Parnas [95], Memon et al. [80], Xie and Memon [134], Pezze and Zhang [96], Barr et al. [7], Segura et al. [104], and Li and Offutt [69].

(iv) Infrastructure for executing the test cases

A testing infrastructure is a system which integrates all required tools and techniques for testing the SUT. It includes generation of the test suite, test oracle, test execution, and search of failure-inducing combinatorial interaction faults. The required infrastructure is often specific to the SUT. The test cases are generally abstract descriptions, and they need to be transformed into concrete test cases that can be executed. The testing infrastructure sets up the environment for executing the test cases. For example, the SUT may need to be initialized and brought up to a ready state before it can be executed. Furthermore, to maintain independence between different test executions, the environment may need to be reset after each execution. For additional information about testing infrastructures see Bertolino [9], and Bonn et al. [11].

(v) Combinatorial fault localization

Combinatorial fault localization refers to the process of identifying from the pass/fail data one or more failure-inducing combinatorial interaction faults which triggered a failure of the SUT. One wants to identify interaction faults involving the least number of factors. Failure-inducing interaction faults can be identified non-adaptively, i.e., using only the pass/fail data of the initial test suite. This approach typically requires the initial test suite to be constructed from special kinds of covering arrays. Alternatively, failure-inducing interaction faults can be identified in an adaptive manner, where new tests are generated to provide additional information that is needed for combinatorial fault localization. The following references discuss various approaches and methods for combinatorial fault localization: Zeller and Hildebrandt [139], Yilmaz et al. [137], Colbourn and McClary [27], Nie and Leung [86], Zhang and Zhang [141], Colbourn and Syrotiuk [28], Seidel et al. [105], Ghandehari et al. [42], Niu et al. [89], and Niu et al. [90].

(vi) Correction of the SUT

The ultimate objective of any software testing is to detect defects in the SUT and correct them. Once one or more failure-inducing combinatorial interaction faults have been identified the underlying defects in the SUT are searched and corrected. Detection and correction of defects requires complete access to the SUT including the source code of software. The following references discuss various approaches and methods for correction of defects: Jones and Harrold [51], Baah et al. [6], Jin and Orso [50], Wong et al. [129], and Pearson et al. [94].

In the literature on software testing the terms ‘defect’ and ‘fault’ are used as synonym. We have used the term ‘fault’ for an ‘interaction fault’ which is a failure-inducing combination of the test values of a small number of factors. Every real application has special features for which the above sub-processes need to be modified appropriately.

6 Comparison of Combinatorial Testing and Factorial Experiments

Combinatorial testing and factorial experiments are both black-box investigations with the following differences.

(i) Objective of investigation

The object of a factorial experimentation is to develop an empirical statistical model for the relationship between the settings (levels) of the important input factors and the output response of a natural or man-made system. Information for treatment comparison and system optimization comes out of the statistical model. The object in combinatorial testing is to detect one or more combinations of the values of a least number of factors for which a software-based system may not behave as expected (correctly). The expected behavior of the SUT for each test case must be pre-determined. When such a combination of values (called a combinatorial interaction fault) is detected, the underlying defect in the software-based system is searched and corrected. In combinatorial testing, statistical models are not used.

(ii) Numbers of factors and test values

Factorial experiments deal with a small number of factors. An investigation may start with, say, ten candidate factors determined from technical knowledge. From these candidates, two to four important factors are selected based on technical knowledge and/or screening factorial experiments. The few important factors are used to develop a statistical model. In a factorial experiment, the factors have a few test values, typically two to four. Combinatorial testing may involve tens, hundreds or thousands of factors (called parameters). A recent CT investigation involved 2,127 factors each having 3, 7 or 10 test values, Smith et al. [116]. In a CT investigation the factors may have many test values. Thus, Colbourn (Website) [24] lists covering array numbers for up to 25 test values per factor. Recently one of our collaborators requested and received strength $t = 3$ covering arrays for 751 factors each having from 6 to 25 test values per factor.

(iii) Replication error

In a factorial experiment, the response may change when a test case is repeated. This change is attributed to experimental replication error. Therefore, one or more test cases are repeated to estimate variation from experimental error. In unreplicated factorial experiments, three-factor and higher order interaction effects are regarded as variation from experimental replication error. In most applications of combinatorial testing, the result of a test case (pass or fail) does not change when the test case is repeated. Thus, each test case needs to be run only once.

(iv) Combinatorial criterion

In a factorial experiment, the combinatorial criterion for developing a design matrix is best estimation of the unknown parameters of the associated statistical model while limiting the total number of test cases. In a fractional factorial experiment, factorial effects are confounded with each other, Thus, 'best estimation' largely means that low order factorial effects are not confounded with each other. In combinatorial testing, the combinatorial criterion for defining a test suite is that for each set of t-factors all possible combinations of the test values (t-tuples) are included in the test suite, and the total number of test cases is minimized for a specified value of the strength t (forbidden combinations are excluded).

(v) Unary factors

In a factorial experiment, every factor has plural test values (levels) because the objective is to compare various alternatives. In combinatorial testing some factors (parameters) may have only one value. Such unary factors are required to determine whether a combination involving their unary values is an interaction fault.

(vi) Conclusion of investigation

In a factorial experiment, the empirical statistical model is often intended for conditions wider than those in which the experiment was done. For example, conclusions from an investigation conducted in an industrial laboratory may be intended for manufacturing conditions or customer's use conditions. The conclusions from combinatorial testing are relevant only for the specific system that was tested and only for the conditions in which it was tested.

7 ACTS/IPOG for Generating Test Suites for CT

ACTS is a collection of research tools for combinatorial testing which is freely available at the NIST Combinatorial testing page (Website) [88]. The main test suite generation tool in ACTS is IPOG. See, Yu et al. [138] for details. Three interfaces are offered: Graphic User Interface (GUI), Command Line Interface (CLI), and Application Programming

Interface (API). The GUI is user-friendly. The CLI and the API are offered to facilitate test automation. The major features of ACTS/IPOG include the following:

(i) Generation of high strength test suites

ACTS/IPOG generates combinatorial t-way test suites for the strength t up to 6, which is considered enough for most CT applications. ACTS/IPOG includes two test generation modes: scratch and extend. The scratch mode allows a test suite to be built from scratch. The extend mode allows a test suite to be built by extending an existing test suite. By extending an existing test suite one can save the effort that has already been spent in the testing process.

(ii) Variable-strength test suites

ACTS/IPOG allows factor (parameter) relations to be created and covered with different strengths in the same test suite. A factor relation is a group of factors that are closely related to each other and whose interactions need to be tested. Factor relations may overlap; that is, a factor may be involved in more than one relation.

(iii) Constraint support

ACTS/IPOG allows the user to specify constraints that must be satisfied for a test case to be valid. The specified constraints are considered in test generation to assure that the resulting test suite will cover, and only cover, combinations that satisfy these constraints. In ACTS/IPOG, constraints are specified using logic expressions and a subset of arithmetic operators that are common in practical applications. ACTS/IPOG includes two algorithms for constraint support. One is based on a constraint solver, and the other is based on forbidden tuples. The latter algorithm is more efficient for constraints that are less complex.

(iv) Negative Testing

Negative testing (or robustness testing) is used to test whether a system handles impermissible inputs correctly. Impermissible values are tested to assure that they are rejected by the system. ACTS/IPOG allows the user to designate some values of a factor (parameter) as impermissible values. During test generation, ACTS/IPOG ensures that each test case contains at most one impermissible value. This is to avoid potential masking effect between impermissible values. In addition, ACTS/IPOG ensures that every impermissible value is combined with every (t - 1)-way combination of permissible values. Currently, ACTS/IPOG supports negative testing only for individual factor-values that are impermissible. That is, it does not support negative testing of impermissible combinations of factor-values where each individual value is permissible.

(v) Coverage Verification

This feature is used to verify whether a test suite satisfies t-way coverage. Coverage of t-way combinations is one way of assuring that an adequate input model has been defined [60]. If no constraint is specified, ACTS/IPOG verifies that all t-way combinations are covered. If some constraints are specified, it verifies that all the t-way combinations satisfying the constraints are covered. A test suite to be verified can be a test suite previously generated by ACTS or a test suite supplied by the user and imported into ACTS/IPOG.

ACTS/IPOG is slow when the strength t, the number of factors (parameters) k, and the number of values v are large. CAgen is a new research tool to generate combinatorial test suites with support for constraints which is very fast and can handle arbitrary strength. The architecture of CAgen is based on the core algorithm implementations described in Kleine and Simos [58]. CAgen algorithms use significantly less memory and can generate test suites for large numbers of factors. CAgen is freely available at SBA-Research/CAgen (Website) [102]. Features of CAgen are described in Wagner et al. [127].

8 Security Testing of Software-Based Systems

Software-based systems are becoming increasingly complex and interconnected. The security of such systems is a critical concern. For example, the so-called Heartbleed bug, which allowed anyone on the Internet to read the memory of systems protected by the OpenSSL software (for example, banking applications), highlighted the critical need to assure attack-free implementations of software-based systems [32].

An important domain for the application of CT is testing for the security of software-based systems. Security testing can be (i) security functional testing or (ii) security vulnerability testing. Security functional testing verifies

that the specified security requirements are implemented correctly. Security vulnerability testing detects existence of inadvertent vulnerabilities. The latter technique uses the simulation of attacks and other kinds of penetration testing attempting to compromise the security of a system by playing the role of an adversary trying to attack the system and exploit its vulnerabilities [5]. Security testing is challenging because modelling of vulnerabilities is specific to the application and the identification of factors triggering such exploits is difficult. Simos et al. [113] outlines a research program to bridge the gap between CT and security testing, and in the process, establish a new research field: combinatorial security testing. Many applications illustrate the effectiveness of CT for security testing, in a wide range of applications ranging from web technologies [39,40], [111,112] and network protocols [113] to operating systems [38,41] and cryptographic applications [57,84].

Combinatorial testing has been used to test implementations of cryptographic hardware systems; that is, hardware implementations of cryptographic algorithms, for example, Advanced Encryption Standard [57]. In this research field, the problem of malicious hardware logic detection, more commonly referred to as the problem of detecting hardware Trojan horses or simply "Trojans" is especially important, where a well-designed Trojan activates under rare conditions and can escape detection during testing [2]. Such conditions cannot be exhaustively searched, especially in the case of cryptographic core implementations with hundreds of inputs. Kitsos et al. [57] explored the applicability of CT to the detection of Trojans that allegedly reside inside embedded chips which perform encryption and decryption (of cryptographic algorithms). It was demonstrated that CT provides the theoretical guarantees for activating a Trojan of specific lengths by covering all input combinations. These findings indicated that CT methods can dramatically improve the existing Trojan detection capabilities by reducing the number of tests needed by several orders of magnitude.

9 Main-Points and Comments for Further Research and Application

Combinatorial testing (CT) is an investigation in which the software-based system under test (SUT) is executed for a suite of test cases to identify one or more combinations of the least number of test values for which the SUT does not behave as expected (that is, correctly). The expected behavior is pre-determined for each test case. A combination of a few test values whose presence in a test case makes the SUT behave different from the expected behavior is called a combinatorial interaction fault. A combinatorial interaction fault indicates the presence of some defect in the SUT. When one or more combinatorial interaction faults are identified, the underlying defects in the SUT are searched and corrected.

Combinatorial testing began with the use of orthogonal arrays (OAs) to detect pairs of values for which the SUT does not behave as expected. Analyses of the reports of actual failures showed that such pairwise testing is useful, but it may not be enough because some defects may show only as combinatorial interaction faults involving three or more values. Also, covering arrays (CAs) are better suited than OAs for combinatorial testing. CAs can be constructed for any number of factors, any numbers of test values, and any strength (while, useful OAs exist only for certain conforming values of the parameters). For a given number of factors, values, and strength, a CA of fewer number of rows than the number of rows in a corresponding OA can usually be constructed. In generating test suites based on CAs, forbidden combinations can be excluded. Also, variable-strength CAs can be used to cut down the number of test cases.

Combinatorial testing and factorial experiments (FE) are both black-box investigations in which the internal workings of the SUT are opaque, unavailable or not referred. But FE and CT have fundamental differences. In a factorial experiment the objective is to form an empirical statistical model for the SUT. The statistical model is in turn used for one or more of the four practical ends: comparison, screening, prediction, and optimization. In combinatorial testing the object is to detect one or more combinations of the values of a least number of factors for which a software-based system may not behave correctly. In CT, statistical models are not used. Factorial experiments deal with a few factors each having a few test values. (When the number of candidates is more-than-a-few, two to four important factors are selected.) Combinatorial testing may involve tens, hundreds or thousands of factors. In CT, the number of test values per factor may be as large as, for example, twenty-five.

For testing a software or a software-based system, no single approach is enough. Plural approaches are generally needed at various stages of software development and installation. Combinatorial testing complements other approaches. Testing can detect the presence of defects, but it cannot assure their absence. Therefore, it is difficult to quantify confidence that a tested system is free of all defects. The defects may range from a minor nuisance to catastrophic.

Combinatorial testing is being increasingly used for investigating the security of software-based systems. Automation is essential for effective software quality assurance [77] and for security testing [130]. CT methods and tools are now well developed to enable automated testing for detection and identification of security flaws, in specific application domains. Also, now it is possible to extend the use of CT to additional application domains (see, [113]). We think combinatorial testing could be useful in the following three applications areas.

(i) Internet of Things (IoT)

Combinatorial methods are ideally suited for an IoT environment, where testing can involve a very large number of nodes and combinations [29,93,126,136]. Combinatorial security testing might prove particularly useful for IoT systems which send and receive data from a large often continually changing set of interacting devices (the number of possible communicating pairs increases with the square of the number of devices).

(ii) Autonomous Systems (AS)

Quality assurance methods and techniques are crucial for safety critical systems like cars, trains, and airplanes. The case of automated driving represents the next level of safety critical systems where additional challenges arise especially in the case of software verification as discussed by Altinger et al. [3] and Wotawa et al. [131]. Combinatorial methods have been employed so far for testing distributed automotive features by Dominka et al. [31], Tao et al. [121], and Li et al. [70]; and for measuring the coverage of test cases used in testing of embedded software in the railway industry as documented by Fifo et al. [33]. However, the added level of complexity when testing for security vulnerabilities for such autonomous systems has not been considered in detail and hence poses a significant research challenge in future applications.

(iii) Artificially Intelligent Software (AI-software)

Verifying the correctness of the implementation of machine learning algorithms like neural networks has become a major topic due to their increasing use in the context of, for example, image recognition, natural language processing, and autonomous systems. Preliminary work by Ma et al. [75], suggests that combinatorial testing and related testing approaches like the ones presented by Chetouane et al. [17] are a promising avenue for testing neural networks and should be explored further. In the case of security, it would be of interest to examine the robustness of implementations of adversarial machine learning algorithms as recently proposed by Biggio and Roli [10].

Acknowledgements Feng Duan helped construct and verify covering arrays. We thank NIST reviewers Brad Alpert, Adam Pintar, and anonymous journal referees.

References

1. ACM Forum on Risks (Website) (http://catless.ncl.ac.uk/Risks/)
2. Adee, S.: The hunt for the kill switch. IEEE Spectrum **45**, 34–39 (2008). https://doi.org/10.1109/MSPEC.2008.4505310
3. Altinger, H., Wotawa, F., Schurius, M.: Testing methods used in the automotive industry: results from a survey. In: Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA 2014), San Jose, CA, USA, July 21, 2014, pp. 1-6 (2014). https://doi.org/10.1145/2631890.2631891
4. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2017)
5. Arkin, B., Stender, S., McGraw, G.: Software penetration testing. IEEE Secur. Privacy **3**, 84–87 (2005). https://doi.org/10.1109/MSP.2005.23
6. Baah, G., Podgurski, A., Harrold, M.: Causal inference for statistical fault localization. In; Proceedings of the 2010 ACM International Symposium on Software Testing and Analysis (ISSTA), Trento, Italy, July 12-16, 2010, pp. 73–84 (2010). https://doi.org/10.1145/1831708.1831717
7. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**, 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785

8. Bell, K.Z., Vouk, M.A.: On effectiveness of pairwise methodology for testing network-centric software, Proceedings of the 2005 International Conference on Information & Communication Technology, Cairo, Egypt, December 5-6, 2005, pp. 221-235 (2005). https://doi.org/10.1109/ITICT.2005.1609626
9. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: Proceedings of Future of Software Engineering (FOSE), Minneapolis, MN, USA, May 23–25 (2007). https://doi.org/10.1109/FOSE.2007.25
10. Biggio, B., Roli, F.: Wild patterns: ten years after the rise of adversarial machine learning. Pattern Recognit. **84**, 317–331 (2018). https://doi.org/10.1016/j.patcog.2018.07.023
11. Bonn, J., Fögen, K., Lichter, H.: A framework for automated combinatorial test generation, execution, and fault characterization. In: Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xian, China, April 23, 2019, pp. 224–233 (2019). https://doi.org/10.1109/ICSTW.2019.00057
12. Box, G.E.P., Hunter, W.G., Hunter, J.S.: Statistics for Experimenters. Wiley, New York (1978)
13. Box, G.E.P., Draper, N.R.: Empirical Model Building and Response Surfaces. Wiley, New York (1987)
14. Brownlie, R., Prowse, J., Phadke, M.S.: Robust testing of AT&T PMX/Starmail using OATS. AT&T Tech. J. **71**, 41–47 (1992). https://doi.org/10.1002/j.1538-7305.1992.tb00164.x
15. Bush, K.A.: Orthogonal arrays of index unity. Ann. Math. Stat. **23**, 426–434 (1952)
16. Chandrasekaran, J., Feng, H., Lei, Y., Kuhn, D.R., Kacker, R.N.: Applying combinatorial testing to data mining algorithms. In: Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Tokyo, Japan, March 13–17, 2017, pp. 253–261 (2017). https://doi.org/10.1109/ICSTW.2017.46
17. Chetouane, N., Klampfl, L., Wotawa, F.: Investigating the effectiveness of mutation testing tools in the context of deep neural networks, In: Rojas, I., Joya, G., Catala, A. (eds.), Advances in Computational Intelligence. IWANN 2019. Lecture Notes in Computer Science, 11506, pp. 766–777. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20521-8_63
18. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**, 193–200 (1994). https://doi.org/10.1049/sej.1994.0025
19. Cochran, W.G., Cox, G.M.: Experimental Designs, 2nd edn. Wiley, New York (1957)
20. Cohen, D.M., Dalal, S.R., Kajla, A., Patton, G.C.: The automatic efficient test generator (AETG) system. In: Proceedings of the 1994 IEEE International Symposium on Software, Reliability Engineering, Monterey, CA, USA, November 6–9, 1994, pp. 303–309 (1994). https://doi.org/10.1109/ISSRE.1994.341392
21. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. **13**, 83–89 (1996). https://doi.org/10.1109/52.536462
22. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. **23**, 437–444 (1997). https://doi.org/10.1109/32.605761
23. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proceedings of the 2003 International Conference on Software Engineering (ICSE), May 3–10, 2003, Portland, OR, USA, pp. 38–48 (2003). https://doi.org/10.1109/ICSE.2003.1201186
24. Colbourn (Website) http://www.public.asu.edu/_ccolbou/src/tabby/catable.html
25. Colbourn, C.J.: Combinatorial aspects of covering arrays. Le Matematiche **59**, 121–167 (2004)
26. Colbourn, C.J., Dinitz, J.H.: The CRC Handbook of Combinatorial Designs, 2nd edn. CRC Press, Boca Raton (2010)
27. Colbourn, C.J., McClary, D.W.: Locating and detecting arrays for interaction faults. J. Comb. Optim. **15**, 17–48 (2008). https://doi.org/10.1007/s10878-007-9082-4
28. Colbourn, C.J., Syrotiuk, V.R.: Coverage, location, detection, and measurement. In: Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Chicago, IL, USA, April 11–15, 2016, pp. 19–25 (2016). https://doi.org/10.1109/ICSTW.2016.38
29. Cui, K., Zhou, K., Qiu, T., Li, M., Yan, L.: A hierarchical combinatorial testing method for smart phone software in wearable IoT systems. Comput. Electr. Eng. **61**, 250–265 (2017). https://doi.org/10.1016/j.compeleceng.2017.06.004
30. Czerwonka (Website) http://www.pairwise.org
31. Dominka, S., Mandl, M., Dubner, M., Ertl, D.: Using combinatorial testing for distributed automotive features: applying combinatorial testing for automated feature-interaction-testing. In: Proceedings of the 2018 IEEE eighth Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, January 8–10, 2018, pp. 490–495 (2018). https://doi.org/10.1109/CCWC.2018.8301632
32. Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of Heartbleed. In: Proceedings of the 2014 ACM Internet Measurement Conference (IMC), Vancouver, Canada, November 5–7, 2014, pp. 475–488 (2014). https://doi.org/10.1145/2663716.2663755
33. Fifo, M., Enoiu, E., Afzal, W.: On measuring combinatorial coverage of manually created test cases for industrial software. In: Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xian, China, April 22–23, 2019, pp. 264–267 (2019). https://doi.org/10.1109/ICSTW.2019.00062
34. Finelli, G.B.: NASA software failure characterization experiments. Reliab. Eng. Syst. Saf. **32**, 155–169 (1991). https://doi.org/10.1016/0951-8320(91)_90052-9
35. Fisher, R.A.: The Design of Experiments. Oliver and Boyd, London (1935)
36. Forbes, M., Lawrence, J.F., Lei, Y., Kacker, R.N., Kuhn, D.R.: Refining the in-parameter-order strategy for constructing covering arrays. J. Res. Natl. Inst. Stand. Technol. **113**, 287–297 (2008)
37. Freedman, D.A.: Statistical Models: Theory and Practice. Cambridge University Press, Cambridge (2005)

38. Garn, B., Simos, D.E.: Eris: A tool for combinatorial testing of the Linux system call interface. In: Proceeding of the 2014 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Cleveland, OH, USA, March 31–April 4, 2014, pp. 58–67 (2014). https://doi.org/10.1109/ICSTW.2014.7
39. Garn, B., Kapsalis, I. Simos, D.E., Winkler, S.: On the applicability of combinatorial testing to web application security testing: a case study. In: Proceedings of the 2014 ACM Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA 2014), San Jose, CA, USA, July 21, 2014, pp. 16–21 (2014). https://doi.org/10.1145/2631890.2631894
40. Garn, B., Simos, D.E., Zauner, S., Kuhn, D.R., Kacker, R.N.: Browser fingerprinting using combinatorial sequence testing. In: Proceedings of the 6-th Annual Symposium on Hot Topics in the Science of Security (HotSoS 2019), Nashville, TN, USA, April 1–3, 2019, Article 7 (2019). https://doi.org/10.1145/3314058.3314062
41. Garn, B., Würfl, F., Simos, D.E.: KERIS: a CT tool of the linux kernel with dynamic memory analysis capabilities. In: Strichman, O., Tzoref-Brill, R. (eds.), Hardware and Software: Verification and Testing. HVC 2017. Lecture Notes in Computer Science, 10629, pp. 225–228. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_17
42. Ghandehari, L., Lei, Y., Kacker, R.N., Kuhn, D.R., Kung, D., Xie, T.: A combinatorial testing-based approach to fault localization. IEEE Trans. Softw. Eng. (TSE) **46**, 616–645 (2019). https://doi.org/10.1109/TSE.2018.2865935
43. Graybill, F.A.: Theory and Application of the Linear Statistical Model. Duxbury Press, North Scituate (1976)
44. Grindal, M., Offutt, J.: Input parameter modeling for combination strategies. In; Proceedings of the 2007 Conference on IASTED International Multi-Conference: Software Engineering, Innsbruck, Austria, February 13–15, 2007, pp. 255–260 (2007). https://dlnext.acm.org/doi/abs/10.5555/1332044.1332085
45. Grindal, M., Offutt, J., Mellin, J.: Managing conflicts when using combination strategies to test software. In: Proceedings of the Australian Software Engineering Conference (ASWEC), Melbourne, Vic., Australia, April 10–13, 2007, pp. 255–264 (2007). https://doi.org/10.1109/ASWEC.2007.27
46. Grochtmann, M., Grimm, K.: Classification trees for partition testing. Softw. Test. Verif. Reliab. **3**, 63–82 (1993). https://doi.org/10.1002/stvr.4370030203
47. Hartman, A., Raskin, L.: Problems and algorithms for covering arrays. Discrete Math. **284**, 149–156 (2004). https://doi.org/10.1016/j.disc.2003.11.029
48. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays: Theory and Applications. Springer, New York (1999)
49. Hinkelmann, K., Kempthorne, O.: Design and Analysis of Experiments, vol. 1. Wiley, New York (1994)
50. Jin, W., Orso, A.: F3: fault localization for field failures. Proceedings of the 2013 ACM International Symposium on Software Testing and Analysis (ISSTA), Lugano, Switzerland, July 15–20, 2013, pp. 213–223 (2013). https://doi.org/10.1145/2483760.2483763
51. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceeding of the 20-th IEEE/ACM International Conference on Automated Software Engineering (ASE), Long Beach, CA, USA, November 07–11, 200, pp. 273–282 (2005). https://doi.org/10.1145/1101908.1101949
52. Kackar, R.N.: Off-line quality control, parameter design, and the Taguchi method (with comments and response). Journal of Quality Technology **17**, 176–209 (1985). https://doi.org/10.1080/00224065.1985.11978964
53. Kacker, R.N., Lagergren, E.S., Filliben, J.J.: Taguchi's orthogonal arrays are classical designs of experiments. J. Res. Natl. Inst. Stand. Technol. **96**, 577–591 (1991)
54. Kampel, L., Simos, D.E.: Set-Based Algorithms for Combinatorial Test Set Generation. In: Wotawa, F., Nica, M., Kushik, N. (eds.), Testing Software and Systems. ICTSS 2016. Lecture Notes in Computer Science, 9976, pp. 231–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47443-4_16
55. Kempthorne, O.: The Design and Analysis of Experiments. Wiley, New York (1952)
56. Khalsa, S., Labiche, Y.: An orchestrated survey of available algorithms and tools for combinatorial testing. In: Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering (ISSRE), Naples, Italy, November 3–6, 2014, pp. 323–334 (2014). https://doi.org/10.1109/ISSRE.2014.15
57. Kitsos, P., Simos, D.E., Torres-Jimenez, J., Voyiatzis, A.G.: Exciting FPGA cryptographic Trojans using combinatorial testing. In; Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE), Gaithersburg, MD, USA, November 2–5, 2015, pp. 69–76 (2015). https://doi.org/10.1109/ISSRE.2015.7381800
58. Kleine, K., Simos, D.E.: An efficient design and implementation of the in-parameter-order algorithm. Math. Comput. Sci. **12**, 51–67 (2018). https://doi.org/10.1007/s11786-017-0326-0
59. Kuhn, D.R., Kacker, R.N., Lei, Y.: Introduction to Combinatorial Testing. CRC Press, Boca Raton (2013)
60. Kuhn, D.R., Kacker, R.N., Lei, Y., Simos, D.E.: Input space coverage matters. IEEE Comput. **53**, 37–44 (2020). https://doi.org/10.1109/MC.2019.2951980
61. Kuhn, D.R., Raunak, M.S., Kacker, R.N.: It doesn't have to be like this: cybersecurity vulnerability trends, IT professional,19 (November/December 2017), pp. 66–70 (2017). https://doi.org/10.1109/MITP.2017.4241462
62. Kuhn, D.R., Reilly, M.J.: An investigation of the applicability of design of experiments to software testing. In: Proceedings of the 27-th NASA/IEEE Software Engineering Workshop, Goddard Space Flight Center, Greenbelt, MD, USA, December 5–6, 2002, pp. 91–95 (2002). https://doi.org/10.1109/SEW.2002.1199454
63. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. **30**, 418–421 (2004). https://doi.org/10.1109/TSE.2004.24

64. Lawrence, J.F., Kacker, R.N., Lei, Y., Kuhn, D.R., Forbes, M.: A survey of binary covering arrays. Electron. J. Comb. 18, Article 84 (2011). https://www.combinatorics.org/ojs/index.php/eljc/article/view/v18i1p84
65. Lei, Y., Tai, K.C.: In-parameter order: a test generation strategy for pairwise testing. In: Proceedings of the 3rd IEEE International High Assurance Systems Engineering Symposium (HASE), Washington, DC, USA, November 13–14, 1998, pp. 254–261 (1998). https://doi.org/10.1109/HASE.1998.731623
66. Lei, Y., Kacker, R., Kuhn, D., Okun, V., Lawrence, J.: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verif. Reliab. **18**, 125–148 (2008). https://doi.org/10.1002/stvr.381
67. Lejay, L.V., Shasha, D.E., Palenchar, P.M., Kouranov, A.Y., Cruikshank, A.A., Chou, M.F., Coruzzi, G.M.: Adaptive combinatorial design to explore large experimental spaces: approach and validation, Systems Biology, 1, pp. 206–212 (2004).https://www.ncbi.nlm.nih.gov/pubmed/17051692
68. Li, D., Hu, L., Gao, R., Wong, W. E., Kuhn, D.R., Kacker, R.N.: Improving MC/DC and fault detection strength using combinatorial testing. In; Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability, and Security (QRS 2017), Prague, Czech Republic, July 25–29, 2017, pp. 297–303 (2017). https://doi.org/10.1109/QRS-C.2017.131
69. Li, N., Offutt, J.: Test oracle strategies for model-based testing. IEEE Trans. Softw. Eng. (TSE) **43**, 372–395 (2017). https://doi.org/10.1109/TSE.2016.2597136
70. Li, Y., Tao, J., Wotawa, F.: Ontology-based test generation for automated and autonomous driving functions. Inf. Softw. Technol. **117**, Article **106200** (2020). https://doi.org/10.1016/j.infsof.2019.106200
71. Li, X., Gao, R., Wong, W.E., Yang, C., Li, D.: Applying combinatorial testing in industrial settings. In: Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), Vienna, Austria, August 1–3, 2016, Article 16378499 (2016). https://doi.org/10.1109/QRS.2016.16
72. Lin, D.K.J., Draper, N.R.: Projection properties of Plackett and Burman designs. Technometrics **34**, 423–428 (1992)
73. Lin, K.M., Kackar, R.N.: Optimizing the wave soldering process. Electronic Packaging & Production, February 1986, pp. 108–115 (1986). Reprinted in Dehnad, K. (eds) Quality Control, Robust Design, and the Taguchi Method, pp. 143–157. Springer, Boston, MA, 1989. https://doi.org/10.1007/978-1-4684-1472-1_7
74. Lott, C., Jain, A., Dalal, S.: Modeling requirements for combinatorial software testing. ACM SIGSOFT Softw. Eng. Notes **30**, 1–7 (2005). https://doi.org/10.1145/1082983.1083281
75. Ma, L., Juefei-Xu, F., Xue, M., Li, B., Li, L., Liu, Y., Zhao, J.: DeepCT: tomographic combinatorial testing for deep learning systems. In: Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, February 24–27, 2019, pp. 614–618 (2019). https://doi.org/10.1109/SANER.2019.8668044
76. Mandl, R.: Orthogonal Latin squares: an application of experiment design to compiler testing. Commun. ACM **28**, 1054–1058 (1985). https://doi.org/10.1145/4372.4375
77. Mariani, L., Hao, D., Subramanyan, R., Zhu, H.: The central role of test automation in software quality assurance. Softw. Qual. J. **25**, 797–802 (2017). https://doi.org/10.1007/s11219-017-9383-5
78. Masuyama, M.: On difference sets for constructing orthogonal arrays of index two and strength two, Reports of Statistical Application Research, 5, pp. 27–34. Union of Japanese Scientists and Engineers (JUSE), Tokyo, Japan (1957)
79. McCulloch, C.E., Searle, S.R.: Generalized, Linear, and Mixed Models. Wiley, New York (2001)
80. Memon, A., Banerjee, I., Nagarajan, A.: What test oracle should i use for effective GUI testing? In; Proceedings of the 18-th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada, October 6–10, 2003, pp. 164–173 (2003). https://doi.org/10.1109/ASE.2003.1240304
81. Mercan, H., Javeed, A., Yilmaz, C.: Flexible combinatorial interaction testing. IEEE Trans. Softw. Eng. (2020). https://doi.org/10.1109/TSE.2020.3010317
82. Miller, R.G.: Beyond ANOVA, Basic and Applied Statistics. Wiley, New York (1986)
83. Montgomery, D.C.: Design and Analysis of Experiments, 3rd edn. Wiley, New York (1991)
84. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R.N.: Finding bugs in cryptographic hash function implementations. IEEE Trans. Reliab. **67**, 870–884 (2018). https://doi.org/10.1109/TR.2018.2847247
85. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys (CSUR), 43, Article 11 (2011). https://doi.org/10.1145/1883612.1883618
86. Nie, C., Leung, H.: The minimal failure-causing schema of combinatorial testing. ACM Trans. Softw. Eng. Methodol. (TOSEM) **20**, Article 15, (2011). https://doi.org/10.1145/2000799.2000801
87. NIST Covering array tables (Website) https://math.nist.gov/coveringarrays/
88. NIST Combinatorial testing page (Website) https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software
89. Niu, X., Nie, C., Lei, Y., Leung, H., Wang, X.: Identifying failure-causing schemas in the presence of multiple faults. IEEE Trans. Softw. Eng. **46**, 141–162 (2020). https://doi.org/10.1109/TSE.2018.2844259
90. Niu, X., Nie, C., Leung, H., Lei, Y., Wang, X., Xu, J., Wang, Y.: An interleaving approach to combinatorial testing and failure-inducing interaction identification. IEEE Trans. Softw. Eng. **46**, 584–615 (2020). https://doi.org/10.1109/TSE.2018.2865772
91. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Commun. ACM **31**, 676–686 (1988). https://doi.org/10.1145/62959.62964
92. Panario, D., Saaltink, M., Stevens, B., Wevrick, D.: An extension of a construction of covering arrays. J. Comb. Des. **28**, 842–861 (2020). https://doi.org/10.1002/jcd.21747
93. Patil, A.H.: Design and implementation of combinatorial testing based test suites for operating systems used for internet of things, Lulu.com (2019)

94. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: Proceedings of the 39th IEEE International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, May 20–28, 2017, pp. 609–620 (2017). https://doi.org/10.1109/ICSE.2017.62
95. Peters, D., Parnas, D.: Using test oracles generated from program documentation. IEEE Trans. Softw. Eng. **24**, 161–173 (1998). https://doi.org/10.1109/32.667877
96. Pezze, M., Zhang, C.: Chapter one—Automated test oracles: a survey. Adv. Comput. **95**, 1–48 (2014). https://doi.org/10.1016/B978-0-12-800160-8.00001-2
97. Plackett, R.L., Burman, J.P.: The design of optimum multifactorial experiments. Biometrika **33**, 305–325 (1946). https://www.jstor.org/stable/pdf/2332195.pdf
98. Raghavarao, D.: Constructions and Combinatorial Problems in Design of Experiments. Wiley, New York (1971)
99. Raktoe, B.L., Hedayat, A.S., Federer, W.T.: Factorial Designs. Wiley, New York (1981)
100. Rao, C.R.: Hypercubes of strength d leading to confounded designs of factorial experiments. Bull. Calc. A Math. Soc. **38**, 67–68 (1946)
101. Sato, S., Shimokawa, H.: Methods for setting software test parameters using the design of experiments method (in Japanese), Proceedings of 4th Symposium on Quality Control in Software, Union of Japanese Scientists and Engineers (JUSE), 1984, pp. 1–8 (1984)
102. SBA-Research/CAgen (Website) https://matris.sba-research.org/tools/cagen
103. Searle, S.R.: Linear Models. Wiley, New York (1971)
104. Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A.: A survey on metamorphic testing. IEEE Trans. Softw. Eng. **42**, 805–824 (2016). https://doi.org/10.1109/TSE.2016.2532875
105. Seidel, S.A., Sarkar, K., Colbourn, C.J., Syrotiuk, V.R.: Separating interaction effects using locating and detecting arrays. In: Iliopoulos, C., Leong, H., Sung, W.K. (eds.) Combinatorial Algorithms. IWOCA 2018. Lecture Notes in Computer Science, 10979, pp. 349–360. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94667-2_29
106. Seiden, E.: On the problem of construction of orthogonal arrays. Ann. Math. Stat. **25**, 151–156 (1954)
107. Shasha, D.E., Kouranov, A.Y., Lejay, L.V., Chou, M.F., Coruzzi, G.M.: Using combinatorial design to study regulation by multiple input signals. A tool for parsimony in the post-genomics era, Plant Physiology **127**, 1590–1594 (2001). http://www.plantphysiol.org/content/127/4/1590
108. Sherwood, G.B.: Effective testing of factor combinations, Proceedings of 3rd International Conference on Software Testing, Analysis and Review (STAR-1994), Washington, DC, USA, May 8–12, 1994, pp. 151–166 (1994). http://testcover.com/pub/background/star1994_paper.pdf
109. Shimokawa, H.: Method of generating software test cases using the experimental design (in Japanese), Report on Software Engineering SIG, Information Processing Society of Japan, No.1984-SE-040 (1985)
110. Simos, D.E., Bozic, J., Garn, B., Leithner, M., Duan, F., Kleine, K., Lei, Y., Wotawa, F.: Testing TLS using planning-based combinatorial methods and execution framework. Softw. Qual. J. **27**, 703–729 (2019). https://doi.org/10.1007/s11219-018-9412-z
111. Simos, D.E., Garn, B., Zivanovic, J., Leithner, M.: Practical combinatorial testing for XSS Detection using Locally Optimized Attack Models. In: Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xian, China, April 22–23, 2019, pp. 122–130 (2019). https://doi.org/10.1109/ICSTW.2019.00040
112. Simos, D.E., Zivanovic, J., Leithner, M.: Automated combinatorial testing for detecting SQL vulnerabilities in web applications. In: Proceedings of the 14th International Workshop on Automation of Software Test, collocated with 41st ACM/IEEE International Conference on Software Engineering (ICSE), Montreal, Canada, May 27, 2019, pp 56–61 (2019). https://doi.org/10.1109/AST.2019.00014
113. Simos, D.E., Kuhn, D.R., Voyiatzis, A., Kacker, R.N.: Combinatorial methods in security testing. IEEE Comput. **49**, 80–83 (2016)
114. Sloane, N.J.A.: (Website) http://neilsloane.com/oadir/index.html
115. Sloane, N.J.A.: Covering arrays and intersecting codes. J. Comb. Des. **1**, 51–63 (1973). https://doi.org/10.1002/jcd.3180010106
116. Smith, R., Jarman, D., Kuhn, D. R., Kacker, R. N., Simos, D., Kampel, L., Leithner, M., Gosney, G.: Applying combinatorial testing to large-scale data processing at Adobe. In: Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xian, China, April 22–23, 2019, pp.190–193 (2019). https://doi.org/10.1109/ICSTW.2019.00051
117. Snedecor, G.W., Cochran, W.G.: Statistical Methods, 7th edn. Iowa State University Press, Ames (1980)
118. Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., Su, Z.: A survey on data-flow testing. ACM Computing Surveys **50**, Article 5 (2017). https://doi.org/10.1145/3020266
119. Taguchi, G.: System of Experimental Design, Vol 1 and 2. Maruzen, Japan (in Japanese) (1977)
120. Tai, K.C., Lei, Y.: A test generation strategy for pairwise testing. IEEE Transactions on Software Engineering **28**, 109–111 (2002). https://doi.org/10.1109/32.979992
121. Tao, J., Li, Y., Wotawa, F., Felbinger, H., Nica, M.: On the industrial application of combinatorial testing for autonomous driving functions. In: Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xian, China, April 22–23, 2019, pp. 234-240 (2019). https://doi.org/10.1109/ICSTW.2019.00058
122. Tatsumi, K.: Test-case design support system. In: Proceedings of the International Conference on Quality Control, Tokyo, Japan, pp. 615–620 (1987)
123. Tatsumi, K., Watanabe, S., Takeuchi, Y., Shimokawa, H.: Conceptual support for test case design. In: Proceedings of 11th IEEE Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, October 1987, pp. 285–290 (1987)

124. Torres-Jimenez (Website) https://www.tamps.cinvestav.mx/~oc/
125. Torres-Jimenez, Izquierdo-Marquez: Survey of covering arrays. In: Proceedings of 2013 IEEE International Symposium and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, September 23–26, 2013, pp. 20–27 (2013). https://doi.org/10.1109/SYNASC.2013.10
126. Voas, J., Kuhn, D.R., Laplante, P.: Testing IoT systems. In: Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), Bamberg, Germany, March 26–29, 2018, pp. 48–52 (2018). https://doi.org/10.1109/SOSE.2018.00015
127. Wagner, M., Kleine, K., Simos, D.E., Kuhn, D.R., Kacker, R.N.: CAGEN: a fast combinatorial test generation tool with support for constraints and higher-index arrays. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, October 24–28, 2020, pp. 191–200 (2020). https://doi.org/10.1109/ICSTW50294.2020.00041
128. Wallace, D.R., Kuhn, D.R.: Failure modes in medical device software: an analysis of 15 years of recall data. Int. J. Reliab. Qual. Saf Eng. **8**, 351–371 (2001). https://doi.org/10.1142/S021853930100058X
129. Wong, W., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**, 707–740 (2016). https://doi.org/10.1109/TSE.2016.2521368
130. Wotawa, F.: On the automation of security testing. In: Proceedings of the 2016 IEEE International Conference on Software Security and Assurance (ICSSA), St. Polten, Austria, August 24–25, 2016, pp. 11–16 (2016). https://doi.org/10.1109/ICSSA.2016.9
131. Wotawa, F., Peischl, B., Klueck, F., Nica, M.: Quality assurance methodologies for automated driving. e & i Elektrotechnik und Informationstechnik **135**, 322–327 (2018). https://doi.org/10.1007/s00502-018-0630-7
132. Wu, C.F., Hamada, M.S.: Experiments: Planning, Analysis, and Optimization, 2nd edn. Wiley, New York (2009)
133. Wu, H., Nie, C., Petke, J., Jia, Y., Harman, M.: Comparative analysis of constraint handling techniques for constrained combinatorial testing. IEEE Transactions on Software Engineering (Early Access) (2019). https://doi.org/10.1109/TSE.2019.2955687
134. Xie, Q., Memon, A.: Designing and comparing automated test oracles for GUI-based software applications. ACM Trans. Softw. Eng. Methodol. (TOSEM) **16**. Article 4 (2007). https://doi.org/10.1145/1189748.1189752
135. Yan, J., Zhang, Jian: A backtracking search tool for constructing combinatorial test suites. J. Syst,. Softw. **81**, 1681–1693 (2008). https://doi.org/10.1016/j.jss.2008.02.034
136. Yang, J., Zhang, H., Fu, J.: A fuzzing framework based on symbolic execution and combinatorial testing. In: Proceedings of the 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, Beijing, China, August 20–23, 2013, pp. 2076-2080 (2013). https://doi.org/10.1109/GreenCom-iThings-CPSCom.2013.389
137. Yilmaz, C., Cohen, M., Porter, A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans. Softw. Eng. (TSE) **32**, 20–34 (2006). https://doi.org/10.1109/TSE.2006.8
138. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: ACTS: a combinatorial test generation tool. In: Proceedings of the 2013 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Luxembourg, March 18–22, 2013, pp. 370–375 (2013). https://doi.org/10.1109/ICST.2013.52
139. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng.(TSE) **28**, 183–200 (2002). https://doi.org/10.1109/32.988498
140. Zhang, Z., Liu, X., Zhang, J.: Combinatorial testing on ID3v2 tags of MP3 files. In: Proceedings of the 2012 IEEE International Conference on Software Testing, Verification and Validation, Workshops (ICSTW), Montreal, QC, Canada, April 17–21, 2012, pp. 587–590 (2012). https://doi.org/10.1109/ICST.2012.145
141. Zhang, Z., Zhang, J.: Characterizing failure-causing parameter interactions by adaptive testing. In; Proceedings of the 2011 ACM International Symposium on Software Testing and Analysis (ISSTA), Toronto, ON, Canada, July 17–21, 2011, pp. 331–341 (2011). https://doi.org/10.1145/2001420.2001460
142. Zhivich, M., Cunnigham, R.: The real cost of software errors. IEEE Privacy Secur. **7**, 87–90 (2009). https://doi.org/10.1109/MSP.2009.56

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.