An Empirical Comparison of Combinatorial and Random Testing

Laleh Sh. Ghandehari[1], Jacek Czerwonka[2], Yu Lei[1], Soheil Shafiee[1], Raghu Kacker[3], Richard Kuhn[3]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas 76019, USA
[2] Microsoft Research, Redmond, Washington 98052, USA
[3]Information Technology Laboratory National Institute of Standards and Technology, Gaithersburg, Maryland 20899, USA

Abstract- **Some conflicting results have been reported on the comparison between t-way combinatorial testing and random testing. In this paper, we report a new study that applies t-way and random testing to the Siemens suite. In particular, we investigate the stability of the two techniques. We measure both code coverage and fault detection effectiveness. Each program in the Siemens suite has a number of faulty versions. In addition, mutation faults are used to better evaluate fault detection effectiveness in terms of both number and diversity of faults. The experimental results show that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better, but with a very small margin. Overall, the differences between the two techniques are not as significant as one would have probably expected. We discuss the practical implications of the results. We believe that more studies are needed to better understand the comparison of the two techniques.**

Keywords- Combinatorial Testing, Random Testing, Software Testing.

I. Introduction

Software failures are often caused by interactions of a few input parameters. A technique called t-way combinatorial testing, or t-way testing, employs a test set that covers all t-way interactions, i.e. interactions that involve no more than t parameters. If parameters and values are modeled correctly, a t-way test set guarantees to expose all failures that involve no more than t parameters. In practical applications, t is typically a small integer that is no more than six [13].

Many empirical studies show that t-way testing can be very effective in fault detection while significantly reducing the number of tests. However, a question that is often asked by the research community is about the comparative effectiveness of t-way testing. That is, how does t-way testing compare to other testing techniques? In particular, how does t-way testing compare to random testing?

Some conflicting results have been reported in the literature. The studies such as [5][8][9][11] find that t-way testing is more effective than random testing. However, other studies such as [1][4][6][7] suggest that there is no significant difference between t-way testing and random testing. This lack of consensus suggests a need for more studies to better understand the effectiveness of these two techniques.

In this paper, we report a new study that responds to the above need. In particular, we investigate the stability of the two testing techniques. For a given test strength t, multiple test sets can be generated to satisfy t-way coverage. Similarly, multiple random test sets of the same size can be generated. The notion of stability refers to the degree to which the effectiveness of such multiple test sets varies. In practice, testers normally execute only one test set that is essentially an arbitrary selection among multiple possible test sets. The more stable a testing technique, the more confidence one has about the effectiveness of the test set that is actually executed.

In our study, we use the Siemens suite as our subject programs. The Siemens suite has been used a benchmark to evaluate the effectiveness of many testing techniques. The suite consists of seven programs, each of which has a number of faulty versions. Our earlier work modeled the input space of these programs [17]. In this current study, for a given test strength t, a total of 100 t-way test sets are generated for each program. For each t-way test set, a random test set of the same size is also generated. Both t-way and random test sets are generated using the same input models in [17].

The effectiveness of an individual test set is measured in terms of code coverage and fault detection. Code coverage data are collected by running test sets on the error-free version of each program. For fault detection, we run test sets on the error-free version and the faulty versions of each program. A fault is detected if the faulty version produces a different output than the error-free version. A mutation test tool called Milu [18] is used to generate additional faulty versions for three programs in the Siemens suite. Mutation faults increase the number and diversity of the faults used in our experiments and thus helps to better evaluate fault detection effectiveness.

The results of our study suggest that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. This can be partially explained by the fact that most random test sets have a high percentage of t-way coverage. That is, while a random test set does not cover all the t-way combinations, it covers most of them. A small number of combinations being missing does not always make a difference on code coverage and fault detection results.

It is important to make several notes about the results of our study. First, we used the same input model for t-way and random testing. While t-way test generation is computationally more expensive than random test

Table I. Characteristics of subject programs

Programs	LOC	Number of functions	Number of faulty versions	Model	Number of constraints
printtokens	472	18	7	$(2^2) \times (2^4) \times (5) \times (8) \times (2 \times 7)^3$ *	4**
				$(4^7 \times 2^2)$	14
printtokens2	399	19	10	$(2^2) \times (2^4) \times (5) \times (8) \times (2 \times 7)^3$ *	4**
				$(4^7 \times 2^2)$	14
replace	512	21	32	$(2^4 \times 4^{16})$	36
schedule	292	18	9	$(2^1 \times 3^8 \times 8^2)$	0
schedule2	301	16	10	$(2^1 \times 3^8 \times 8^2)$	0
tcas	141	9	41	$(2^7 \times 3^2 \times 4^1 \times 10^2)$	0
totinfo	440	7	23	$(3^3 \times 5^2 \times 6^1)$	0

* The model of the *replace* program has two levels; sub level consists of 7 sub models and the top model with 9 parameters. Three out of 7 sub models share the same model, two parameters with 2 and 7 values.

** The second sub model with (2^4) input model, has 4 constraints and the other does not have any constraints.

generation, both procedures are automated. Thus the advantage of random testing in terms of reducing test generation cost is not as significant in practice as one would probably have perceived. Second, in our experiments, the size of a random test set is set to be the same as its corresponding t-way test. However, when we apply random testing in practice, we need to decide when to stop, i.e., how many tests are sufficient. This can be a difficult decision. In this respect, t-way testing has an advantage in that it has a well-defined stopping point, i.e., achieving full t-way coverage. Finally, we must acknowledge that our study is limited in terms of both the number and sizes of the subject programs, and the number and types of faults. More studies are needed to obtain a better understanding.

The remainder of this paper is organized as follows. In section II, we describe our experimental design. Section III reports experimental results. Section IV provides some general discussion about the experimental results. Section V describes threats to validity. Section VI gives an overview of work that is related to ours. Section VII provides concluding remark.

II. Experimental Design

This section describes the design of our experiments, including the subject programs, the evaluation metrics, and the test generation procedure used by our experiments.

A. Subject Programs

Our experiments use the Siemens suite from the Software Infrastructure Repository [16]. This suite contains 7 programs. Two programs, *printtokens* and *printtokens2*, have the same specification but different implementations. They tokenize a text file and determine the type of each token. The *replace* program takes three inputs, *pattern*, *substitute* and *input text*, and it *replace*s every match of *pattern* in *input text* with *substitute*. Two programs, *schedule* and *schedule2*, provide two different implementations of a scheduling scheme that determines the execution order of a set of processes based on their priorities. The *tcas* program is an aircraft collision avoidance system. The *totinfo* program takes as input a file containing one or more tables, and computes the total degree of freedom and chi-square of rows and columns.

In the Siemens suite, each program has an error-free version and several faulty versions. There also exists a test set for each program. These test sets are not used in our experiments. Table I shows some characteristics of the subject programs. The second column shows the number of lines of (uncommented) code. The third column shows the number of functions. The fourth column shows the number of faulty versions. The fifth column shows the input models used for test generation. The input models are shown in an exponential format. For example, *totinfo* has six parameters, where three, two and one of them have a domain size of 3, 5 and 6, respectively. The model of this program is shown in an exponential format by $(3^3 \times 5^2 \times 6^1)$. The last column shows the number of constraints in the input model. The details of the models are explained in [17].

In addition to the faulty versions that come with the Siemens suite, a mutation testing tool called Milu [18] is used to generate additional faulty versions. This helps to better evaluate fault detection effectiveness both in terms of number and diversity of faults. The number of mutants generated by Milu is typically large, and running hundreds of test sets over them is very time consuming. In our experiments, we select three programs, *replace*, *schedule* and *totinfo*, and for each of the three programs, we select a few functions, for mutant generation.

We refer to faults in the faulty versions provided by the Siemens suite as Siemens faults, and faults that are generated by mutation as mutation faults.

Table II shows some characteristics of generated mutants. The second column shows the number of functions selected for each program. Note that *schedule* is smaller than the other two programs, the mutants are generated for the entire program. The third column indicates the number of mutants generated. The fourth column shows that the number of terminating mutants that are used by our

Table II Characteristics of generated mutants

Programs	Number of functions used for mutants generation	Number of mutants	Number of terminating mutants
replace	4	143	128
schedule	18	94	93
totinfo	2	151	149

experiments. Mutants that did not terminate after 1 minute were excluded in our experiments.[1])

We do not select *printtokens* and *printtokens2* for mutant generation because of the hierarchical nature of their input models. We do not select *schedule2* since it has the same model as *schedule*. Also *tcas* is not selected because it has complex decision logic and its mutants are likely to represent faults with strength of more than 6.

B. Evaluation Metrics

We measure the effectiveness of an individual test set in two dimensions, i.e., code coverage and fault detection.

For code coverage, line and branch coverage collected for each test set run with the error-free version of each program. A tool called *gcov* is used to gather coverage data. The tool is executed with the "branch-probabilities" option, and the "line executed" output is taken for line coverage and the "taken at least once" output is used for branch coverage.

For fault detection, we check how many faults can be detected by a test set. A fault is detected if the output of a faulty version is different from the output of the error-free version by one or more tests in a test set.

For code coverage and fault detection data collected from a group of test sets, we compute minimum, first quartile, median, third quartile, maximum, spread and relative standard deviation. The first five measures summarize the effectiveness of the test sets as a group, whereas the latter two summarize how stable the results are across different test sets in the group.

C. Test Generation

For each subject program, we generate 100 t-way test sets for each strength t, where t is from 2 to 5. There are a total of 400 t-way test sets for each program. We use PICT [20] to generate t-way test sets. PICT uses a greedy, random algorithm for t-way test generation and allows the user to specify a seed. In order to obtain different test sets, a different seed is given each time a test set is generated. Test sets are compared to ensure that no two test sets are exactly the same. In our experiments no redundant test sets are detected.

For *replace*, we did not generate 5-way test sets as they are very large, and take too much time to execute. On average, there are 12604.22 tests in a 5-way test set for *replace* and it takes 3.22 seconds to execute each test (against all the 32 faulty versions in the Siemens suite). Thus it takes about 11.27 hours to execute each test set. The time needed to execute 100 test sets is prohibitive and thus we did not conduct 5-way testing for *replace* in our experiments. Note that our experiments are conducted on a PC that has a Pentium (R) 4 (2.40 GHZ) processor and 2 GB memory and that runs Ubuntu 12.04 LTS (32bit).

For each t-way test set, we generate a random test set of the same size. The same input model used by t-way test generation is used for random test generation. If the input model of a program does not have any constraint, a random test is generated by simply giving each parameter a random value of its domain. Otherwise, additional care needs to be taken to ensure that all the constraints are satisfied. More details about random test generation with the presence of constraints can be found in [17].

III. EXPERIMENTAL RESULTS

In this section, we first present the test generation results, i.e., some important properties and statistics of the test sets generated in our experiments. Then we present the test execution results in terms of code coverage and fault detection that are achieved by these test sets.

A. Test generation result

Table III shows some statistics about the sizes of the generated test sets including minimum, maximum, average

Table III Test sets' size

Program	Strength	Min	Max	Average	RelStdDev
printtokens	2	42	47	44.46	2.72
	3	113	127	119.6	2.17
	4	307	330	319.97	1.64
	5	763	791	776.38	0.80
replace	2	200	220	210.86	2.18
	3	904	955	928.66	1.10
	4	3730	3805	3773.07	0.41
schedule	2	64	64	64	0
	3	244	259	251.22	1.45
	4	1060	1088	1075.30	0.57
	5	3788	3806	3812.26	0.26
tcas	2	100	100	100	0
	3	400	409	403.38	0.47
	4	1401	1447	1423.28	0.65
	5	4240	4321	4277.85	0.36
totinfo	2	31	35	32.41	3.10
	3	150	158	153.26	0.92
	4	532	560	544.5	1.05
	5	1554	1613	1586.35	0.72

[1] In retrospect, this exclusion is not necessary. Instead, non-terminating mutants should be considered killed. Due to insufficient time we were not able to re-do the experiments by the time of this submission. If this paper is accepted, we will re-do the experiments without this exclusion.

Table IV Combinatorial coverage of random sets

Program	Strength	Min	Max	Average	RelStdDev
printtokens	2	52.94	82.58	72.03	7.88
	3	54.29	88.54	76.95	11.45
	4	61.10	94.27	86.31	9.24
	5	73.68	95.31	91.76	4.13
replace	2	89.38	96.06	94.85	0.95
	3	89.27	96.41	94.46	1.73
schedule	2	91.15	96.53	93.64	1.06
	3	92.51	94.07	93.43	0.39
	4	94.85	95.68	95.30	0.17
	5	95.66	96.19	95.89	0.08
tcas	2	92.23	96.18	94.25	0.74
	3	93.51	95.19	94.30	0.34
	4	95.15	96.00	95.52	0.17
	5	96.05	96.45	96.26	0.08
totinfo	2	75.78	88.67	82.64	2.96
	3	83.18	88.86	86.20	1.31
	4	83.47	87.15	85.05	0.79
	5	81.92	83.73	82.96	0.46

and relative standard deviation. Note that *printtokens* and *printtokens2* use the same input model and thus have the same test sets, and so do *schedule* and *schedule2*. Also note that *printtokens* and *printtokens2* have a hierarchical input model. Due to limited space, we only show statistics for the test sets generated from the top model.

Table IV shows the statistics of the t-way coverage achieved by the random test sets. The t-way coverage of a test set is computed using the ACTS tool with a special option on the command line interface [19]. For most cases, more than 80% (on average) of t-way coverage is achieved by a random test set. The exceptions are for *printtokens* with t = 2 and 3, where the average t-way coverage is more than 70% but lower than 80%. ACTS was not able to compute the t-way coverage for *replace* when t = 4. The reason is that *replace* has a relative large and complex input model while the option for computing t-way coverage in ACTS is mainly experimental and is thus not optimized.

Table V Maximum line and branch coverage results

Programs	Max of line coverage	Max of branch coverage
printtokens	(46.15) (46.67) (45.13) (43.08) (74.36) (35.38) (47.69)	(35.78) (36.7) (38.53) (40.37) (57.8) (27.52) (35.78)
	69.74	55.05
printtokens2	(58.5) (58.5) (57) (71) (73.5) (56.5) (74.5)	(45.68) (46.91) (46.3) (58.02) (67.9) (40.74) (70.99)
	80.5	76.54
replace	88.93	80.56
schedule	94.74	80.30
schedule2	94.57	75
tcas	89.23	90.91*
totinfo	92.68	84.09

* In this program maximum branch coverage is greater than maximum line coverage, the reason is that || and && operators (in an if statement) introduce new branches, in gcov.

B. *Test execution result*

The test execution results are presented in three parts, including code coverage results, Siemens fault detection results, and mutation fault detection results.

Code Coverage: Code coverage is collected by running each test set on the error-free version of each subject program. Table V shows the maximum line and branch coverage achieved by these test sets. Maximum coverage indicates to certain degree the quality of the input model. For *printtokens* and *printtokens2*, the maximum line and

Table VI Coverage results of *replace*

Metric	Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	T-way Testing	88.84	88.84	88.84	88.84	88.84	0	0
		Random Testing	88.84	88.84	88.84	88.84	88.84	0	0
	3	T-way Testing	88.84	88.84	88.84	88.84	88.84	0	0
		Random Testing	88.84	88.84	88.84	88.84	88.84	0	0
	4	T-way Testing	88.84	88.84	88.84	88.84	88.84	0	0
		Random Testing	88.84	88.84	88.84	88.84	88.84	0	0
Branch Coverage	2	T-way Testing	79.44	79.44	79.44	80.56	80.56	1.12	0.69
		Random Testing	79.44	79.44	79.44	80.56	80.56	1.12	0.70
	3	T-way Testing	79.44	80.56	80.56	80.56	80.56	1.12	0.39
		Random Testing	79.44	80.56	80.56	80.56	80.56	1.12	0.24
	4	T-way Testing	80.56	80.56	80.56	80.56	80.56	0	0
		Random Testing	80.56	80.56	80.56	80.56	80.56	0	0

Table VII Coverage results of *schedule*

Metric	Strength	Method	Min	Q1	Median	Q3	Max	spread	RelStdDev
Line Coverage	2	T-way Testing	94.74	94.74	94.74	94.74	94.74	0	0
		Random Testing	94.74	94.74	94.74	94.74	94.74	0	0
	3	T-way Testing	94.74	94.74	94.74	94.74	94.74	0	0
		Random Testing	94.74	94.74	94.74	94.74	94.74	0	0
	4	T-way Testing	94.74	94.74	94.74	94.74	94.74	0	0
		Random Testing	94.74	94.74	94.74	94.74	94.74	0	0
	5	T-way Testing	94.74	94.74	94.74	94.74	94.74	0	0
		Random Testing	94.74	94.74	94.74	94.74	94.74	0	0
Branch Coverage	2	T-way Testing	80.3	80.3	80.3	80.3	80.3	0	0
		Random Testing	80.3	80.3	80.3	80.3	80.3	0	0
	3	T-way Testing	80.3	80.3	80.3	80.3	80.3	0	0
		Random Testing	80.3	80.3	80.3	80.3	80.3	0	0
	4	T-way Testing	80.3	80.3	80.3	80.3	80.3	0	0
		Random Testing	80.3	80.3	80.3	80.3	80.3	0	0
	5	T-way Testing	80.3	80.3	80.3	80.3	80.3	0	0
		Random Testing	80.3	80.3	80.3	80.3	80.3	0	0

Table VIII Coverage results of *totinfo*

Metric	Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
Line Coverage	2	T-way Testing	26.83	28.46	41.46	73.17	86.99	60.16	41.37
		Random Testing	26.83	26.83	38.21	73.17	86.99	60.16	47.17
	3	T-way Testing	39.84	77.24	78.86	89.23	92.68	52.84	17.23
		Random Testing	28.46	75.61	77.24	88.62	92.68	64.22	15.61
	4	T-way Testing	78.86	92.68	92.68	92.68	92.68	13.82	5.47
		Random Testing	77.24	78.86	92.68	92.68	92.68	15.44	6.94
	5	T-way Testing	92.68	92.68	92.68	92.68	92.68	0	0
		Random Testing	78.86	92.68	92.68	92.68	92.68	13.82	1.49
Branch Coverage	2	T-way Testing	27.27	30.68	42.615	68.18	77.27	50	34.92
		Random Testing	27.27	29.55	38.64	67.05	77.27	50	39.32
	3	T-way Testing	39.77	72.73	75	80.11	84.09	44.32	14.78
		Random Testing	30.68	71.59	72.73	78.41	84.09	53.41	13.36
	4	T-way Testing	75	84.09	84.09	84.09	84.09	9.09	3.94
		Random Testing	72.73	75	84.09	84.09	84.09	11.36	5.03
	5	T-way Testing	84.09	84.09	84.09	84.09	84.09	0	0
		Random Testing	75	84.09	84.09	84.09	84.09	9.09	1.08

branch coverage are shown for the top model and all the sub-models. The maximum line and branch coverage achieved by t-way and random test sets are the same. This is consistent with the fact that both types of test set use the same input model.

Tables VI, VII and VIII show some statistics of the code coverage results for three programs, *replace*, *schedule* and *totinfo*. Due to limited space, we do not show statistics for the other programs, which can be found in [21]. These three programs are included since they are selected for mutation testing. Orange cells show cases where random testing performs better than t-way testing.

For the *replace* program, t-way and random testing produce the same results for line coverage. However, when t = 3, random testing has a slightly smaller relative standard deviation for branch coverage than t-way testing (Table VI). For the *totinfo* program, t-way testing outperforms random testing in many cases. For example the minimum

line and branch coverage of t-way testing are greater, sometimes significantly greater, than random testing for t = 3, 4, and 5 (Table VII). When t = 3, random testing has a smaller standard deviation than t-way testing for both line and branch coverage. However t-way testing has higher *min*, *Q1*, *median* and *Q2* than random testing. For the *schedule* programs, t-way and random testing have exactly the same statistics (Table VII).

For t-way test sets, *spread* and *standard deviation* are non - increasing as t increases. This indicates that as t increases, code coverage becomes more stable for t-way test sets. This is true for the other four programs whose results are not shown in the paper. This is, however, not true for random test sets. For example, for *totinfo* (Table VIII), the spreads of both line and branch coverage when t = 3 are greater than when t =2.

Note on programs whose results are not shown here: For *schedule2* and *printtokens*, t-way and random testing had exactly the same results. For *printtokens2*, t-way testing had better result than random testing in all cases. In addition, t-way test sets achieved the maximum coverage when t = 2, while random test sets achieve the maximum coverage when t = 4. For *tcas*, random testing performed better than t-way testing when t = 2, while t-way testing performed better when t > 2.

Siemens Faults: Each program has a number of faulty versions in the Siemens suite in SIR [16]. Table IX show the maximum number of faults that are detected by the t-way and random test sets generated in our experiments. For *printtokens* and *printtokens2*, the results are shown for the top model and all the sub-models. The faults detected by the different models may overlap. For example, the fifth sub-model detected faulty versions 1, 2, 4, and 7, whereas the sixth and seventh sub-models detected faulty versions 4 and 7. However, all the faulty versions are killed by all the models together.

Table IX Maximum number of Siemens faults detected

Programs	Total	Max number of faults detected
printtokens	7	2
printtokens2	10	7
replace	32	32
schedule	9	7
schedule2	10	3
tcas	41	41
totinfo	23	12

Tables X, XI, and XII show the results for three programs, i.e., *replace*, *schedule* and *totinfo* respectively. Again, due to limited space, the results for the other programs are not shown. Instead, they are made available in [21]. Also, orange cells show cases where random test sets have better results than t-way testing.

Table X Siemens faults detection of *replace*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	16	16	18	32	32	16	32.34
	Random Testing	16	18	32	32	32	16	25.46
3	T-way Testing	18	32	32	32	32	14	8.61
	Random Testing	18	32	32	32	32	14	6.21
4	T-way Testing	32	32	32	32	32	0	0
	Random Testing	32	32	32	32	32	0	0

Table XI Siemens fault detection of *schedule*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	7	7	7	7	7	0	0
	Random Testing	7	7	7	7	7	0	0
3	T-way Testing	7	7	7	7	7	0	0
	Random Testing	7	7	7	7	7	0	0
4	T-way Testing	7	7	7	7	7	0	0
	Random Testing	7	7	7	7	7	0	0
5	T-way Testing	7	7	7	7	7	0	0
	Random Testing	7	7	7	7	7	0	0

Table XII Siemens faults detection of *totinfo*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	1	3	5	6	11	10	45.84
	Random Testing	1	3	3	6	9	8	48.35
3	T-way Testing	4	9	10	11	12	8	19.88
	Random Testing	4	9	10	11	12	8	17.97
4	T-way Testing	10	12	12	12	12	2	4.02
	Random Testing	9	12	12	12	12	3	7.04
5	T-way Testing	12	12	12	12	12	0	0
	Random Testing	12	12	12	12	12	0	0

For *replace*, random testing has better *Q1*, *Median*, and *RelStdDev* when t = 2, and better *RelStdDev* when t = 3 (Table X). For *schedule*, random and t-way testing have exactly the same results (Table XI). For *totinfo*, when t = 2, random testing has a smaller *Spread* but it has a higher *Median* and *Maximum*. When t = 3, random testing has a smaller *RelStdDev*, but all the other measures are the same. When t = 4, t-way testing clearly outperforms random testing, and when t = 5 both reach the maximum results.

For both t-way and random test sets, s*pread and RelStdDev* are non-increasing as t increases. This suggests that the fault detection results become more stable as t increases.

Note on programs whose results are not shown here: For *printtokens2* and *schedule2*, t-way and random testing have exactly the same results. For *printtokens*, t-way and random testing have exactly the same results in all the models except for the second sub-model t-way testing outperforms random testing. For *tcas*, random testing performs better when t = 2 and 3, whereas t-way testing performs better when t = 4 and 5. For *tcas*, the fault detection results do not become stable as t increases. The reason is probably because the degree of all the faults in *tcas* is more than 5 [17].

Mutation Faults: Only the first 30 (out of 100) test sets are executed on each mutant. This is because running all test sets for each mutant is prohibitively time consuming. For example, it takes 13.19 hours to execute (and evaluate) a 4-way test set on all the 128 mutants of the *replace* program.

Table XIII show the maximum number of mutants detected by t-way and random testing. For the *replace* program all 128 mutants are detected. For *schedule* and *totinfo*, 22 and 27 mutants could not be detected, respectively.

Tables XIV, XV, and XVI show some statistics of mutation fault detection for *replace*, *schedule* and *totinfo*, respectively. For *replace*, t-way testing performed as good as or better than random testing when t = 2 and 3, whereas

Table XIII Maximum number of mutation faults detected

Programs	Total	Max number of faults detected
replace	128	128
schedule	93	71
totinfo	149	122

random testing performed better when t = 4. More discussion on the latter case is discussed later. For *schedule*, t-way and random testing have the same results. For *totinfo*, when t = 2, t-way testing have better results in all measures except for *Q3* and Max. When t = 3, random testing seems to perform better as it has better results in *Min*, *Q1*, *Spread*, and *RelStdDev*. However, when t = 4, t-way testing clearly outperforms, and also it reaches the maximum point where the maximum number of faults are detected by all test sets. When t = 5, both t-way and random testing reach the maximum point.

For *replace*, when t = 4, the minimum number of faults detected by a t-way test is 26 less than that by a random test set. We randomly selected 4 out of these 26 mutants and analyzed their degrees of faults.

Our investigation showed that all these faults are more than 9-way, i.e., they involve more than 9 parameters. Whereas the probability is not high, we conjecture that the reason why there exists a t-way test set that detects none of these 26 mutants is because this test set does contain any combinations that trigger these higher-degree faults. In contrast, it happens to be that all the random tests happen to contain at least one triggering combination for each of these 26 mutants.

IV. Discussion

In most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. As shown in Table IV, random test sets provided on average a very

Table XIV Mutation faults detection of *replace*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	101	101	101	101	128	27	7.94
	Random Testing	101	101	101	101	128	27	7.94
3	T-way Testing	101	102	127.5	128	128	27	10.63
	Random Testing	101	101	127	128	128	27	11.54
4	Combinatorial testing	101	128	128	128	128	27	3.88
	Random Testing	127	128	128	128	128	1	0.20

Table XV Mutation faults detection of *schedule*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	71	71	71	71	71	0	0
	Random Testing	71	71	71	71	71	0	0
3	T-way Testing	71	71	71	71	71	0	0
	Random Testing	71	71	71	71	71	0	0
4	T-way Testing	71	71	71	71	71	0	0
	Random Testing	71	71	71	71	71	0	0
5	T-way Testing	71	71	71	71	71	0	0
	Random Testing	71	71	71	71	71	0	0

Table XVI Mutation faults detection of *totinfo*

Strength	Method	Min	Q1	Median	Q3	Max	Spread	RelStdDev
2	T-way Testing	61	63	67	68	117	56	25.32
	Random Testing	61	63	66	100	121	60	29.02
3	T-way Testing	63	114	121	122	122	54	17.30
	Random Testing	73	115	121	122	122	49	7.80
4	T-way Testing	122	122	122	122	122	0	0
	Random Testing	115	122	122	122	122	7	1.46
5	T-way Testing	122	122	122	122	122	0	0
	Random Testing	122	122	122	122	122	0	0

high level of combinatorial coverage, almost always in excess of 80% and frequently over 95%. All test sets have some degree of t-way coverage, regardless of how they are generated. The fact that the randomly generated tests had a very high level of t-way coverage can partially explain why there is little difference between the two techniques.

Although there was little difference between combinatorial and random tests at a particular interaction strength t, fault detection increased rapidly with increasing t. For practical testing, the results suggest that higher levels of combinatorial coverage significantly improve fault detection, regardless of whether the combinatorial coverage is produced by t-way or random test generation,

A t-way test set covers all t-way combinations and thus guarantees to detect all t-way faults. Moreover, a t-way test set also covers many combinations whose size is greater than t. Thus, a t-way test set may also detect faults of higher strength, but without guarantee. This phenomenon has been observed in our experiments. For example, all the faults that come with *totinfo* in the Siemens suite have a degree of at least three. However, one of the 2-way test sets generated in our experiments was able to detect 11 of these faults. Another example is the *tcas* program, for which all of the Siemens faults are more than 5-way. Five 5-way test sets generated in our experiments detected all these faults.

V. Threats to validity

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. We have tried to automate the experimental procedure as much as possible, as an effort to remove human errors. In particular, we build a tool that automatically compares the results of the error-free version and a faulty version to evaluate each test run. Further, the consistency of the results are checked manually to determine whether the tool works correctly or not.

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from the Siemens suite [16]; these programs are created by a third party, but the subject programs are programs of relatively small size and with a small number of seeded faults. To mitigate this threat, the mutation faults are added to the experiments. But more experiments on larger programs with real faults can further reduce this threat.

VI. Related work

A number of studies have been reported that evaluates the effectiveness of t-way testing. In this section, we focus on related work that compares the effectiveness of t-way and random testing.

Schroeder et al. in [1] conducted an experiment to compare the fault detection effectiveness of combinatorial and random test sets. The subject programs are two real-life programs in C++, including the Data Management Analysis System (DMAS) and the Loan Arranger System (LAS), only one functionality of each program is tested. Their results show that there is no significant difference in t-way and random testing in terms of fault detection.

DMAS and LAS have 8.7 and 6.2 KLOC, and their input models are represented as $(2^{16} \times 5 \times 8)$ and $(2^{7} \times 3^{10} \times 4^{2})$, respectively. For each program, and for each strength t, where t is from 1 to 4, 10 t-way test sets are generated using a tool called TVG [1]. For each t-way test set, a random test set of the same size is generated. Mutants are created manually to generate faulty versions. Mutants that are killed by all the 1-way sets are removed. A total of 88 mutants for DMAS and a total of 82 mutants for LAS are used in their experiments.

In [1], a random test is generated by randomly selecting a test from all possible tests. This is different from our approach in which a random test is generated by giving each parameter a random value in its domain. This difference may slightly affect the combinatorial test coverage achieved by a random test set. Note that the approach in [1] assumes that all possible tests are first generated, which may not be practical for large input models.

Ellimis et al. [8][10] report an experiment that tests 10 different functions of a system called Wallace that controls a large industrial engine. A mutation tool is used to generate faulty versions. For each function, three test sets are generated, one t-way test set, one pure random test set, and one manually generated test set. Pure random tests are generated without using an input model.

Their results show that 2-way test sets are not as effective as manually generated tests in term of fault detection. But a t-way test set of a higher strength could be as effective as a manually generated test set. Their results also show that random test sets may often provide good results. For example, for 5 out of 10 programs, random and t-way test sets provide the same results, and in one case random test sets even produce better results than t-way test sets.

Several studies are reported that compare t-way testing and random testing for testing logical expressions [6][7][11]. The logical expressions are either taken from a program such as *TCAS* II or generated randomly. Mutants are generated to create faulty versions. The results consistently show that t-way testing is always more effective, and sometimes significantly more, than random testing

Kuhn et al. [5] report a study that applies t-way testing and random testing to detect deadlocks in a network simulator called Simured. The input model for the simulator is $(2^3 \times 3 \times 4^9 \times 5)$. T-way test sets are generated by ACTS with t = 2 to 4. For each t-way test set, eight random test sets of the same size are generated corresponding to each combinatorial set. Their experiments show that (1) random testing has better results than 2-way testing; (2) no significant difference exists between random and 3-way testing; (3) 4-way testing is more effective than random testing.

Bell and Vouk reported applied 2-way testing and random testing to a network-centric software [9]. They found that 2-way testing is more effective in fault detection. In particular, when there is at least one parameter with more than 10 values, random testing does not detect about 75% of faults that are detected by 2-way testing

Bryce et al. compared the coverage of combinatorial and random testing on a system called Flight Guidance System (FGS) [2]. The FGS system has 40 input parameters, each of which has 2 values. Four t-way test sets with t = 2 to 5, as well four random test sets of the same size, are generated. Their results show that t-way testing is more effective than random testing for the FGS system.

Finally we note that Czerwonka reported a study [3], that applies t-way testing to four utility program in Windows 7, including attrib.exe, fc.exe, find.exe and findstr.exe. The focus of the study is to investigate the stability of t-way testing in terms of line and branch coverage. The results show that t-way test sets provide stable coverage when t >= 2. This study, however, does not make a comparison with random testing.

VII. Conclusion

In this paper, we report a study that compares the effectiveness of t-way testing to that of random testing in terms of both code coverage and fault detection. In particular, we investigated the stability of the two techniques. Our results show that in most cases, t-way testing performed as good as or better than random testing. There are few cases where random testing performed better, but with a very small margin. Overall, the differences between the two are not as significant as one would have probably expected. A possible explanation is that most random test sets seem to achieve a high level of t-way

coverage. More studies are needed to better understand the effectiveness of the two testing techniques.

We plan to conduct more empirical studies to further evaluate the effectiveness and stability of combinatorial testing. We plan to use programs that are larger and/or more complex than the Siemens programs. We also plan to conduct studies where the degree of fault can be better controlled. This will help us to better study the relationship between the combinatorial coverage of a test set and the faults the test set is able to detect.

VIII. References

1. P.J. Schroeder, P. Bolaki, V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," In *Proceeding of the International Symposium on Empirical Software Engineering*, pp.49,59, 2004.
2. R.C. Bryce, A. Rajan, M.P.E. Heimdahl, "Interaction Testing in Model-Based Development: Effect on Model-Coverage," In *Proceeding of the 13th Asia Pacific Software Engineering Conference*, pp.259,268, 2006.
3. J. Czerwonka, "On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Deigns", In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 257-266, 2013.
4. J. Bach, P. J. Schroeder, "Pairwise testing: A best practice that isn't", In *Proceeding of the 22nd Annual Pacific Northwest Software Quality Conference*, pp. 180-196, 2004.
5. D.R. Kuhn, R. Kacker, Y.Lei. "Random vs. Combinatorial Methods for Discrete Event Simulation of a Grid Computer Network", In *Proceedings of ModSim World*, pp. 83-88, 2009.
6. S. Vilkomir, O. Starov and R. Bhambroo, "Evaluation of t-wise approach for testing logical expression in Software", In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp.249-256, 2013.
7. W. A. Ballance, S. Vilkomir, W. Jenkins, "Effectiveness of Pair-Wise Testing for Software with Boolean Inputs", In *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp.580-586, 2012.
8. M. Ellims , D. Ince, M. Petre, "The effectiveness of t-way test data generation", In *Proceedings of the 27th international conference on Computer Safety, Reliability, and Security*, pp. 16-29, 2008.
9. K. Z. Bell, M. A. Vouk, "On effectiveness of pairwise methodology for testing network-centric software". In *Proceeding of the ITI 3rd International Conference on Information and Communications Technology*, pp.221,235, 2005
10. M. Ellims, D. Ince, and M. Petre, "AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation," Technical Report 2007/08, Dept. Computer Science, Open University, Milton Keynes, June 2007.
11. N. Kobayashi, T. Tsuchiya, T. Kikuno, "Applicability of non-specification-based approaches to logic testing for software", In *Proceeding of the International Conference on Dependable Systems and Networks*,pp. 337-346, 2001.
12. D. R. Wallace, D. R. Kuhn, "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data", In *Proceeding of the ACS/ IEEE International Conference on Computer Systems and Applications*, pp. 301-311, 2001.
13. D.R. Kuhn, D.R. Wallace, and A.M. Gallo. "Software Fault Interactions and Implications for Software Testing", IEEE Transaction on Software Engineering 30(6):418-421, 2004.
14. M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and D. R. Kuhn, "T-way testing of ACTS: A Case Study", In *Proceedings of the IEEE fifth International Conference on Software Testing, Verification and Validation*, pp.591-600, 2012.
15. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pp. 285-294, 1999.
16. H. Do, S. Elbaum, and G. Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact", Empirical Software Engineering. 10(4):405-435, 2005.
17. L. S. Ghandehari, M. N. Bourazjany, Y. Lei, R.N. Kacker and D.R. Kuhn, "Applying T-way testing to the Siemens Suite", In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 362-371, 2013.
18. Y. Jia and M. Harman. "Milu: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language", TAIC PART '08. Testing: Academic & Industrial Conference, pp. 94-98, 2008.
19. Advanced T-way testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, 2013.
20. J. Czerwonka, "Pairwise testing in real world. Practical extensions to test case generators", In *Proceedings of 24th Pacific Northwest Software Quality Conference*, pp. 419-430, 2006.
21. http://barbie.uta.edu/~laleh/BEN/ben.html