Abstract

BELL, KERA ZAKIYAH. Optimizing Effectiveness and Efficiency of Software Testing: A Hybrid Approach. (Under the direction of Dr. Mladen A. Vouk.)

The overall goal of software testing is to disclose defects efficiently (i.e. as little time and cost as possible) and effectively (i.e. find as many faults as possible). It takes time to understand what to test, to generate test cases, and to execute the test suite. It also takes time to analyze the test results. In a situation where one can parameterize inputs and variables of interest, the cost of generating random operational profile conformant test cases may be acceptable. It is typically O ($M \times p$) where p is the number of parameters and M is the number of test cases where a test case is a vector of p values. However, random testing can result in very large test suites that may take long to execute. Unless an automated oracle is available, results also may take long to analyze. In contrast, systematic approaches tend to generate smaller test suites, thus reducing run-time and analysis costs, but may take much longer to generate since they may require a higher-level of initial expertise to develop. There may also be differences in the ability of various methods to detect faults in the software. A question of interest is the following: Is there a way to take advantage of the benefits of both statistical and systematic approaches in a hybrid scheme to optimize both efficiency and effectiveness?

Hybrid approaches combine one or more testing techniques. This work discusses the basics of the underlying theory and presents the results of the associated experiments and simulations. The specific parameter-based systematic technique is called N-wise testing. Test case vectors are selected without replacement in N-wise testing. The N-wise technique assumes that most faults of interest will be found when all or most of the involved parameter N-tuple values are covered by the tests. If faults involve no more than N parameter values (called N-way faults), then all faults will

be detected. In contrast, random testing is usually driven by operational profile considerations. Test cases are sampled with replacement, and coverage of N-way faults is random. Thus it is not guaranteed that all N-way faults will be detected using random testing. However, a complete N-wise test suite of M test cases guarantees that all N-way defects will be found. Given M random test cases and M N-wise test cases, effectiveness of N-wise testing may be larger than that of random testing. On the other hand, random testing may be more cost-efficient. The efficiency and effectiveness of an approach that combines random testing with N-wise testing is explored.

In the case where all defects are N-way but one could only afford to perform k-wise testing ($k < N$), a hybrid of k-wise systematic testing and random testing may help maximize efficiency and increase effectiveness beyond that offered by either technique on its own. This hybrid approach seems most effective when the minimum number of interacting parameters, N, required to expose a defect is large and the average number of values associated with the p parameters is also large. A potential use of hybrid testing is in testing for faults from failures that may result from a complex combination of interacting parametric values, such as those found in security failures, and in testing highly complex network-based systems and workflows in general.

OPTIMIZING EFFECTIVENESS AND EFFICIENCY OF SOFTWARE TESTING: A HYBRID APPROACH

by

KERA ZAKIYAH BELL

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER SCIENCE

Raleigh

2006

Approved By:

Dr. Donald Bitzer	Dr. Winser Alexander
Dr. Mladen A. Vouk Chair of Advisory Committee	Dr. Laurie Williams

To my mother, who helps make it all possible.

Biography

Kera will be the first of many family members to earn a Ph.D. She received an M. S. in Computer Science from Clark Atlanta University and a B. S. in Mathematics from Spelman College. Kera aspires to encourage young and old alike through life and love. She will embark upon a new career as Assistant Professor of Computer Sciences at Georgia Southern University in Statesboro, Georgia with the love of her life.

Acknowledgements

Thanks to Dr. M. Vouk for daring to believe in me. Thanks to the NASA Harriett G. Jenkins Predoctoral Fellowship Program for making a believer out of me. Thanks Reesy, for being such an incredible friend and "partner in crime" supporting me during the easiest and hardest of times (rhymed by accident :-). Thanks to two of my greatest friends - Richard and Keegan. Thanks Trina, my matron of honor and best friend of all time, for dreaming beyond the horizon. Steven, Kemi, Tu, Shauna, Nneka, Kim, Erin, Desiree . . . I don't know what I'd do without y'all. Here's the proof that even the impossible is possible!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software has proliferated over the years and has become more diverse, complex, and more recently network-based. Consumers indirectly or directly expect a certain quality of the software they use - whether in cars or airplanes they ride, at a check-out line where they make purchases, or when preparing their bank statements. To achieve quality that meets expectations, software must be tested. Properly testing software, especially critical software, is imperative. As one of the verification and validation methods, testing can help prevent catastrophic events and costly losses. This requirement is magnified in missions where software must operate for long periods of time and in environments where direct human intervention is not possible. There are a number of publications that discuss the present state of software testing [37, 84, 38, 39, 43, 22, 21, 55]. Different authors define software testing differently, but the consensus seems to include two things:

1. Software testing is a necessity to help attain any desired level of software quality [21, 50, 63, 43, 47].

2. The goal of software testing is to find problems (i.e. defects) in software so they can be addressed or fixed [22, 21, 63, 43, 47].

Finding software faults and defects before someone else (preferably before the release of that software) helps to prevent potential accidental or malicious exploitation of the faults. Testing is also used to pinpoint sources of software behavior(s) that perform contrary to what is expected. [1]

[1] In this text: a human error or mistake leads to introduction of a physical fault(s) into a software artifact. These initial fault(s) can propagate to the executable code (as physical defects). When these defects are encountered during execution of the code (and given the right operational profile circumstances, environment, combination of parameter values, etc.) they may give rise to an internal error-state. This state may or may not persist but if (perhaps in combination with other error-states) it results in an externally visible anomaly (e.g., unexpected behavior) then we say that a failure has occurred.

Figure 1.1: The Objective of Software Testing is to "Break" Software

Unfortunately, it is quite possible that expected behaviors may still be exploited in some way, which may not necessarily be based on any computer-related malevolence. While software testing is moving from an art to a scientific process, it still has a long way to go [58, 55, 63]. More and more testing is driven by well-defined processes based on statistical and systematic testing approaches and automation is often a part of the solution. However, a lot of ad hoc nature and art remains when manual testing is used. The next several sections review some of the more common software testing approaches.

1.1 Software Testing Techniques

1.1.1 White-Box

White-box testing focuses on the internals of a system (also known as structural information). For example, one may exercise all possible branches or all assignment statements found in source code [34]. White-box testing often involves measurement of the extent of testing through "coverage" of the internal structures. When these structures are related to the faults and defects, this type of testing can provide test-stopping criteria and by name it is considered "good testing" [17, 50, 45, 62]. In practice, this type of testing seems to be most useful when structural information is not overwhelming and is readily available.

Unfortunately, there is no guarantee that the necessary structural information, such as that

found in source code, is always available to a tester. Furthermore, a particular testing strategy may not always cover all faults of interest. In addition, existing white-box testing techniques tend to be time-consuming for large source codes [31]. While a good number of testing faults could probably be detected using white-box techniques, the amount of time needed to dedicate to this type of testing may be outside the constraints allotted to some particular software in practice [66].

1.1.2 Black-Box

Black-box testing, sometimes called functional testing, is concerned with inputs and associated outputs of a system without regard to the internal structure of the system. In this context, a very useful concept is the operational profile of software under consideration [55]. An operational profile, in its simplest form, can be captured by the volume and relative frequency of usage of externally exposed software functions, operations, input variables, parameters, values, and states. An issue that may arise is how often and for what purpose may black-box test-driven sampling of the input space re-sample certain input states. In other words, black-box testing could result in a certain amount of redundancy in functional and input usage. In general practice, reuse of input states occurs and it is known as sampling of input spaces with replacement. During testing this may be inefficient It may take an exponential amount of exposure of software to testing or test operations (e.g., in terms of the number of test cases or in time) to reach a particular level of reliability that an end-user of software may require [66]. While any number of changes can impact and change the operational profile for which the software was designed, anticipatory testing for the related software input parameter interactions becomes more challenging as software becomes more network-based, because distributed communications and networks can expose the software to totally new operational environments (and interactions) very rapidly and in an unpredictable way. New testing considerations may be required to ensure that testing remains sufficient as well as efficient. [2]

1.2 Categories

In this work we will group testing methods into one of the following two categories: systematic and statistical (or random) methods. Systematic testing is based on some set of guiding rules and a knowledge of structural and behavioral aspects of the software that are then translated into a targeted (often) non-random sampling of the software input space. On the other hand, random

[2] Most testers will use a combination of both white-box (structural) and black-box (functional) testing, in a practice sometimes known as gray-box testing, to achieve a certain level of quality [50].

Table 1.1: Schools of Thought of Systematic Testing Versus Random Testing

	Random Testing		
	Expendable	Essential	
Expendable	[redacted]	[12]	**Systematic Testing**
Essential	[58, 21]	[76, 50, 54]	

testing, as its name implies, uses random number generators to produce input data according to a pre-specified profile. For example, uniform random testing will sample all input values with equal weight, while operational profile based testing would sample the input space proportional to its usage in the field. Random testing is a software testing tradition.

It is possible to think of "classical" random testing as the use of minimal knowledge of an input domain to randomly select values for testing [80, 26, 61]. Minimal information may be environmental settings, input names, associated values, etc. On the other hand, according to some researchers, most systematic approaches can be considered instances of partition testing [61, 26]. "Classical" partition testing separates an input domain into disjoint sets or sub-domains based on certain criteria [5, 26, 61]. Each input of the sub-domain shares some specified trait(s) in common. In practice, there may be blurring of systematic methods and random methods when more complex operational-profile based testing is used.

There have been many debates over how to choose a particular partition testing approach and how partition testing compares with random testing. Many of these debates arise when a group of researchers believes random testing is not a good approach. Some researchers believe random testing is expendable, while systematic testing is essential and vice versa. Table 1.1 [34, 50, 5] cross-tabulates some of the papers found in different opinion domains.

It is not unusual to find a focus on statistical methods to help deal with test generation state space explosion issues [15, 55, 34]. The most common problem with statistical testing is that a large number of test cases may have to be generated before an adequate level of testing has been reached. While uniform random testing is the simplest approach, risk-tuned field-representative operational distributions are probably the most appropriate way of generating random test cases [55]. They tend to offer a summarized collection of information less detailed than that used for white-box testing but gives more internal information than black-box testing alone.

As already mentioned, most systematic techniques can be categorized as instances of partition testing [61]. Some authors categorize a systematic technique by the group of tools it uses; others define the actual technique by the process that is complemented by the appropriate set of tools [54]. Another issue is step-by-step repeatability of the process.

Preferences vary. Some like systematic techniques alone, and some prefer a combination of both systematic and statistical techniques [55, 21]. Recent trends seem to lean towards approaches such as chaining, combinatorial approaches, counterexamples, behavior and data modelling, orthogonal arrays, control flow graphs, and others [68, 19, 53, 85, 17, 30, 29, 11, 69, 67]. For purposes of this discussion, systematic testing will be limited to combinatorial approaches.

1.3 Challenges in Software Testing

There are several major challenges that arise with testing modern software. Some are as follows:

- automation of the test case generation and their execution
- development of domain and software engineering expertise needed for adequate testing
- formalization and modelling of the software specifications and implementations, and software testing process and effects.
- the growing complexity of the modern software-based systems.

Good software testing is determined by the defined processes and their accompanying automated tools (if required), as well as the faults testing is able to discover, not the tools alone [63, 50, 54]. However, for introduction of new test generation techniques, it is important for automation to be possible as well. Manual testing can be cumbersome because it is tedious, time-consuming, and error prone [47, 21, 50, 62, 43].

On the other hand, some critical activities may require manual testing [50, 21]. Also, functions may only need to be tested once, and may therefore benefit from manual testing in terms of cost effectiveness [50, 54]. A tester should be cautioned since it is difficult to precisely repeat a manual test scenario. [21, 54]. Human error can become a factor if repetition is required [63].

Unfortunately, even if processes are well-defined and the tools give all the information required, testers may lack the expertise and experience to follow those processes or use those tools effectively [21, 63]. Perry has stated that "...less than one percent of software professionals have been formally trained in testing techniques" [63]. There may be domain knowledge issues as well. It is therefore imperative that automated tools subsume parts of the process so that the overall process becomes as simple and as robust as possible to compensate for lack of experience.

Formal modelling is sometimes used to verify and validate software, and aid in testing through formal test cases and graphical representations. Models can take many forms: coverage models, control flow models, finite state machine models, etc. As software increases in complexity, such models, depending on the level of abstraction and detail used, may be a less realistic representation of a real system or may be difficult to understand. Because large systems usually have a large number of states, state explosion problem may restrict or hinder formal modelling in practice [32]. In practice there could be (and usually are) unexpected differences, states and scenarios not accounted for by the models [77].

System and software complexity has dramatically increased as the hardware becomes more capable, as functionalities and expectations of customers grow, and the world becomes more networked. For example, network-based environments are usually nondeterministic since more than one path can be taken to achieve a particular outcome. Although "traditional" software may have a single entry point for input and/or data entered into a system serially, that often does not hold true for a large networking environment. For example, Figure 1.2 illustrates a simple network with three possible data entry points indicated by the arrows. Traditionally, one may consider one data entry point at a time. In a network setting, however, it is possible for two inputs or even all three to occur simultaneously. This adds to the complexity of the software under test in this environment.

While techniques for testing network-based software may involve a series of syntax-oriented operations, in actual operation there may be a number of things that can go wrong that have very little to do with input parameter syntax faults, but a lot to do with complex interactions among the input and environmental parameters. Therefore, it is important to extend testing philosophy to more complex notions of semantics and correlated events and operations. For instance, a number of security violations can take place through memory abuse - often the result of a time-correlated series of unexpected or improperly tested operations or events, through correlated disturbance of established connections - such as spoofing [87]. Unfortunately, some of the catalysts for these problems may not be known a priori [77].

1.4 Random Testing

1.4.1 Advantages

There are many advantages to random testing. It seems to perform favorably well in acceptance testing and system testing phases of the software process [80]. It is often simple to

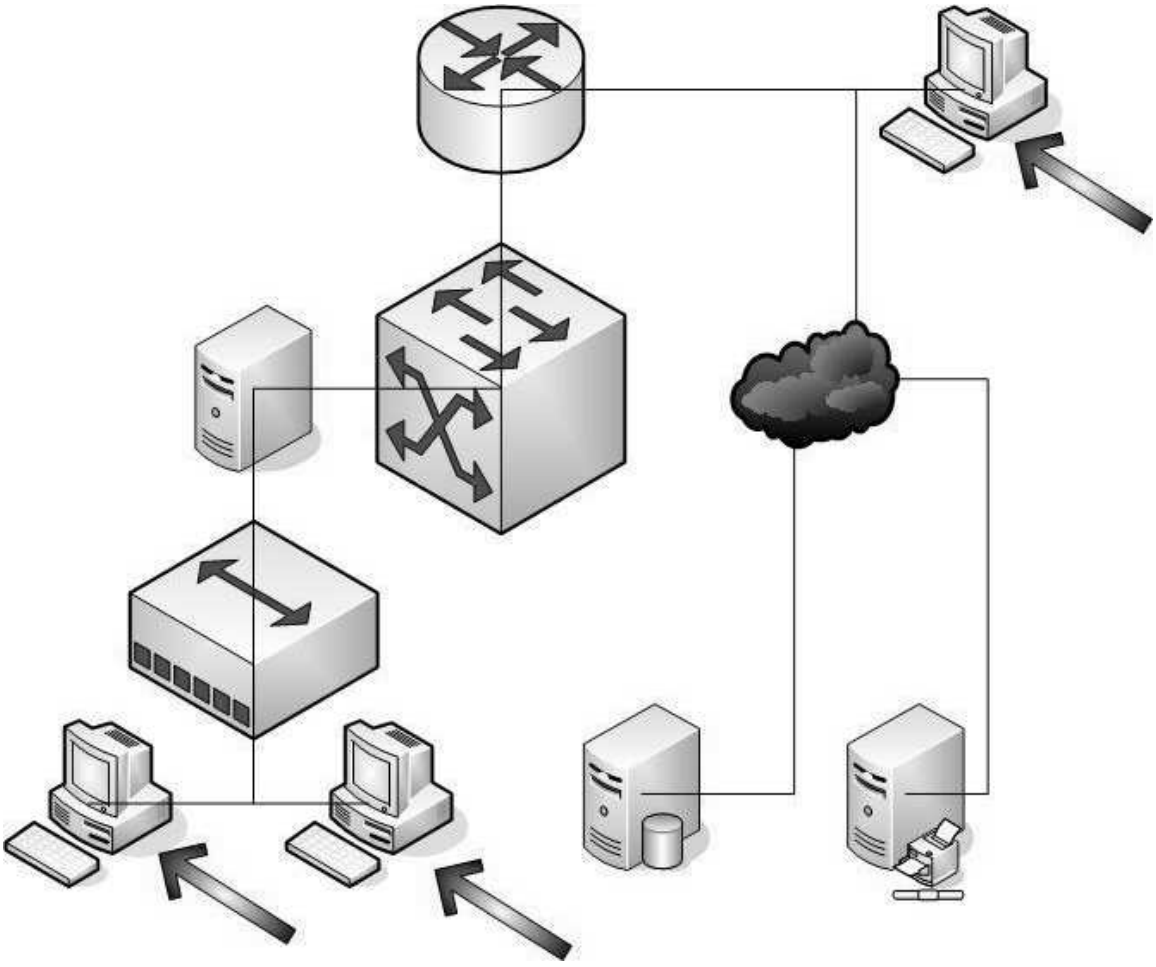

Figure 1.2: Example of a Small Network Adding Complexity to a Testing - Note the Figure Input Points are Marked A1, A2, A3

implement and can generate a complete test suite rather quickly. Typically, in $O(v \times M)$ for random testing with replacement where v is the total number of values in the space and M is the total number of test cases or $O(p \times M^2)$ for random testing without replacement where p is the number of parameters and M is the number of test cases. According to Lewis, random testing can sometimes catch defects that systematic processes could possibly miss [50], especially when operational profile distribution is used in testing [55].

1.4.2 Issues

Random testing is sometimes unfavorable because it does not take the structure of a program into account [26]. Also, unless random testing is tied to the operational profile, there is no

direct measure of reliability [21, 55]. Furthermore, some researchers feel that less systematic strategies tend to have lower confidence bounds [80, 21], and random testing could result in wasted effort. One reason is that in practice random testing is often done with replacement so that there are repeated test cases [21]. It also may take time to ascertain correctness of the output (the "oracle" issue) and pinpoint the source of a problem using this technique. Craig and his colleagues claim that random testing offers little in terms of reliability since risk cannot be directly measured and coverage is not well-defined [21]. Some employ brute force testing based on uniform distributions while others have used more weighted distributions, such as an operational profile [55]. Another issue with random testing is that, unless a tester knows the precise random generating functions used, repeatability may become an issue which could affect potential automated activities [50, 21, 54]. Random testing may suffer from similar problems manual testing does. A decrease in automation could mean a decrease in effectiveness [54]. However, some supporters of systematic testing argue that these semi-random techniques do not provide additional function coverage over systematic techniques [21].

It has been reported that random testing alone may find, "...only one out of every three errors present in the software," [54] and those errors may be part of that 1/3 defects that systematic processes may miss [50]. It seems that very few studies would suggest performing random testing alone to effectively catch defects [54, 61, 50].

On the other hand, there are authors who believe that reliability can effectively be improved and estimated using random testing [12, 68, 55, 80].

1.4.3 An Example

Suppose that the testing space shown in Figure 1.2 can be expressed as a collection of parameters and their associated values based on the data entry points indicated by the arrows. Formally, this space reading left-to-right can be represented as A_1, A_2 and A_3 where $1 \leq i \leq 3$ and A_i is an input parameter.

Let Table 1.2 represent a list of parameters (A_1, A_2, A_3) with their associated values listed underneath. Each subscript is the associated parameter and each sub-subscript represents an enumerated value for that particular parameter. Assume that an 'off' state exists for every parameter. Without loss of generality, we can assume that the first value represents an 'off' state (i.e. a_{i_0} is off).

In this example, if all possible values are executed one at a time, then there would be 11

Table 1.2: Example of Parameters and Associated Values

$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
a_{1_0}	a_{2_0}	a_{3_0}
a_{1_1}	a_{2_1}	$*a_{3_1}$
	a_{2_2}	a_{3_2}
		a_{3_3}
		a_{3_4}
		a_{3_5}

unique executions possible. Suppose a defect could be detected when a_{1_1} is executed or when a_{3_1} is executed (indicated by the '*'). There is a 2/11 chance of finding the indicated defects in one trial. How many trials would minimally be required to attain a probability of more than 1/2? Probability is f = 1-2/11 of not detecting the problem in one attempt. Assuming that the space is sampled with replacement (which in this case is really not the best option), and (assuming independence of the trials) f to the M^{th} power is the probability of not detecting a problem in M trials. Hence, solving Equation 1.1 for M will compute the expected minimum, M_{min} (Equation 1.2). In this case, at least 4 test cases would need to be executed to hit at least one of the defects with probability of 1/2. If a 99.99% chance of finding at least one defect is desired, then one would need to execute at least 46 test cases. However, in this case, exhaustive testing means the execution of all single values, which would be 11 test cases. Obviously, sampling with replacement is very inefficient since executing just 11 test cases (without replacement) would find all problems. As the input space grows (in terms of a large number of parameters and values), the situation may change.

$$1 - \left(1 - \frac{2}{11}\right)^M > \frac{1}{2} \tag{1.1}$$

$$M_{min} = \left\lfloor \frac{\log\left(1 - \frac{1}{2}\right)}{\log\left(1 - \frac{2}{11}\right)} + 1 \right\rfloor \tag{1.2}$$

1.5 Combinatorial Testing

Single-input executions were discussed in the Section 1.4.3 example. There are situations in which a number of parameters interact and can be (and should be) tested simultaneously. [3] An example of a system exhibiting this behavior can be seen in Figure 1.3. In this situation, there are a number of combinations among input parameters that can exist. Assume that one parameter can take

[3] Data that can be tested simultaneously may sometimes be referred to as parallel data.

exactly one value at any given instant. [4] Generating test cases based on combinations of parameter values is commonly referred to as combinatorial testing. Exhaustively testing everything, (i.e., all combinations) often leads to a combinatorial (state) explosion.

For example, let a system have 20 parameters and 5 possible values for each, then there would be 5^{20} or 95,367,431,640,625 possible combinations. To put this into perspective, if one test case takes 1 μs to test, then the entire suite would take approximately 3 years to execute! This may be an acceptable time frame for some systems, but in other situations - such as rapid testing - testing needs to be completed within months if not weeks [22].

There is ample research on this type of testing and how to intelligently reduce the number of tests that arise in a combinatorial environment [12, 68, 19, 17, 86, 85, 90, 53, 49, 8, 7, 9, 10]. One such systematic technique is called N-wise testing. The randomly generated counterpart of N-wise testing is called random combinatorial testing [68]. In random combinatorial testing at least two inputs are tested simultaneously.

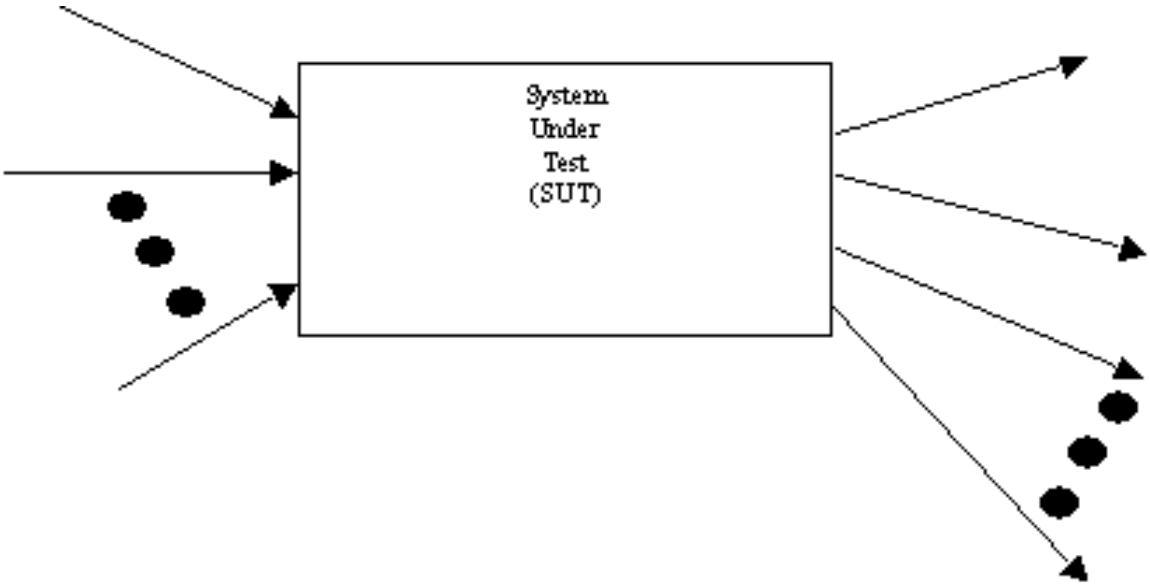

Figure 1.3: Example of a System with Many Inputs which can be Tested Simultaneously

[4] If a parameter can assume more than one value at a time, that parameter can be split into more than one parameter until it can assume only one value in an instantaneous moment.

1.5.1 N-wise Testing

N-wise testing is a systematic technique which ensures that all possible combinations of the values of N parameters appear in a test suite at least once. N is a positive integer in the range $2 \leq N \leq p$ where p is the total number of parameters of the system under test. [5] N-wise testing is generally used in cases where systems have at least 2 parameters and at least 2 values per parameter. This testing guarantees that all possible N-tuples are covered in the test suite at least once [17, 75].

Naturally, N can vary but cannot exceed the total number of parameters. N-wise testing has been shown to be NP-Complete [49], so algorithms performing this type of testing are based on heuristics, and there is much research dedicated to deciding how to perform N-wise testing [17, 85, 49, 53, 75].

1.5.2 Advantages and Disadvantages

An advantage of combinatorial testing is that it is designed to meet the needs of systems with a number of parallel inputs. This type of testing can result in a relatively small test suite, especially if the N-way combinations are sampled without replacement. Several studies have found this to be effective [19, 12, 68]. Pairwise testing (and N-wise testing in general) has been shown to be fairly effective with some types of software systems and may prove beneficial for testing other types of systems [19]. An unfortunate recent example of a software-related loss, which might have been prevented by exploring relationships among different input and environmental parameters related to the operation of the system as a whole, is the case of the NASA Mars Climate Orbiter. It crashed onto the surface of Mars because of a mismatch in the measurement units assumed for input data and those expected by the navigational software. The estimated loss was $165 million dollars [40]. This failure was a direct result of inadequate relationship interaction testing of the system parameters.

One of the pitfalls of performing combinatorial testing is the classification of parameters. Such an activity may be relatively easy when performing configuration testing where the parameters are pre-defined. However, in complex systems deciding to group certain behaviors together is a non-trivial task. Although most researchers have attempted to make such groupings disjoint, it has been suggested that this may not be possible [70]. Another problem is that although non-exhaustive combinatorial testing, such as pairwise testing, has been shown effective with some systems, there may be situations in which one may not choose to use this technique. [72] Smith et al. reported that

[5] When $N = 2$, it is known as the special case called pairwise testing.

many critical defects apparently missed in one of their systems using pairwise testing. Hence, there is a need to join combinatorial testing with other techniques to provide a more complete coverage of the software fault-space.

1.5.3 Example

A test suite can be represented as a single parameter, two parameters, or all parameters. Let a_{i_j} represent the j^{th} value of parameter A_i. In general, the exhaustive combinatorial test suite could formally be represented as a Cartesian product $\{(A_1 \times A_2 \times A_3)\}$ of all possible values as shown in Equation 1.3. Based on the example given in Table 1.2, there are a total of $2 \times 3 \times 6 = 36$ possible combinations when all parameters are considered to interact with each other. One possible combination is $(a_{1_0}, a_{2_1}, a_{3_5})$. All possible combinations are shown in Table 1.3.

$$Exhaustive\ Suite = \{(a_{1_{i_1}}, a_{2_{i_2}}, a_{3_{i_3}}) : (1 \leq i_1 \leq |A_1|), (1 \leq i_2 \leq |A_2|), (1 \leq i_3 \leq |A_3|)\} \tag{1.3}$$

Consider the example in Table 1.3, which could also be considered 3-wise exhaustive testing. Table 1.4 shows a test suite that would satisfy $N = 2$ or pairwise testing. Table 1.4 is smaller since there are 18 test cases total; and all that is required is that each pair of parameters appear at least once in the suite. Notice that there could be more than one possible pairwise test suite.

Suppose a failure can be seen when test cases # 11 and # 26 are executed. There is a 2/36 (or 1/18) chance of finding a defect in one trial. How many trials would minimally be required to attain a probability of more than 0.5 of detecting the problems? If sampling is in the discrete space without replacement, this probability is determined by the hypergeometric distribution. If sampling is with replacement it is determined by the binomial distribution (e.g., [65]). For random testing with replacement, Equation 1.4 was solved for M and the expected minimum was computed expressed as M_{min} which can be seen in Equation 1.5. In this case, a minimum of 13 test cases would need to be executed to expect to hit at least one of the defects with probability of at least 1/2. If 99.99% chance of finding at least one defect is desired, then execute at least 162 test cases. Again, this value is interesting when one considers that there are only 36 test cases possible in the exhaustive test suite. That means that many redundant tests would be performed. Although the random testing would still be effective, the efficiency would be compromised with the introduction

Table 1.3: Exhaustive Combinatorial Test Suite: All Possible Combinations of Parameters and Values

Test Index	$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
1	a_{1_0}	a_{2_0}	a_{3_0}
2	a_{1_0}	a_{2_0}	a_{3_1}
3	a_{1_0}	a_{2_0}	a_{3_2}
4	a_{1_0}	a_{2_0}	a_{3_3}
5	a_{1_0}	a_{2_0}	a_{3_4}
6	a_{1_0}	a_{2_0}	a_{3_5}
7	a_{1_0}	a_{2_1}	a_{3_0}
8	a_{1_0}	a_{2_1}	a_{3_1}
9	a_{1_0}	a_{2_1}	a_{3_2}
10	a_{1_0}	a_{2_1}	a_{3_3}
*** 11**	a_{1_0}	a_{2_1}	a_{3_4}
12	a_{1_0}	a_{2_1}	a_{3_5}
13	a_{1_0}	a_{2_2}	a_{3_0}
14	a_{1_0}	a_{2_2}	a_{3_1}
15	a_{1_0}	a_{2_2}	a_{3_2}
16	a_{1_0}	a_{2_2}	a_{3_3}
17	a_{1_0}	a_{2_2}	a_{3_4}
18	a_{1_0}	a_{2_2}	a_{3_5}
19	a_{1_1}	a_{2_0}	a_{3_0}
20	a_{1_1}	a_{2_0}	a_{3_1}
21	a_{1_1}	a_{2_0}	a_{3_2}
22	a_{1_1}	a_{2_0}	a_{3_3}
23	a_{1_1}	a_{2_0}	a_{3_4}
24	a_{1_1}	a_{2_0}	a_{3_5}
25	a_{1_1}	a_{2_1}	a_{3_0}
*** 26**	a_{1_1}	a_{2_1}	a_{3_1}
27	a_{1_1}	a_{2_1}	a_{3_2}
28	a_{1_1}	a_{2_1}	a_{3_3}
29	a_{1_1}	a_{2_1}	a_{3_4}
30	a_{1_1}	a_{2_1}	a_{3_5}
31	a_{1_1}	a_{2_2}	a_{3_0}
32	a_{1_1}	a_{2_2}	a_{3_1}
33	a_{1_1}	a_{2_2}	a_{3_2}
34	a_{1_1}	a_{2_2}	a_{3_3}
35	a_{1_1}	a_{2_2}	a_{3_4}
36	a_{1_1}	a_{2_2}	a_{3_5}

Table 1.4: Pairwise Test Suite: All Possible Pairs of Parameters and Values

Test Index	$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
1	a_{1_0}	a_{2_0}	a_{3_0}
8	a_{1_0}	a_{2_1}	a_{3_1}
15	a_{1_0}	a_{2_2}	a_{3_2}
22	a_{1_1}	a_{2_0}	a_{3_3}
29	a_{1_1}	a_{2_1}	a_{3_4}
36	a_{1_1}	a_{2_2}	a_{3_5}
10	a_{1_0}	a_{2_1}	a_{3_3}
35	a_{1_1}	a_{2_2}	a_{3_4}
6	a_{1_0}	a_{2_0}	a_{3_5}
34	a_{1_1}	a_{2_2}	a_{3_3}
5	a_{1_0}	a_{2_0}	a_{3_4}
30	a_{1_1}	a_{2_1}	a_{3_5}
2	a_{1_0}	a_{2_0}	a_{3_1}
21	a_{1_1}	a_{2_0}	a_{3_2}
25	a_{1_1}	a_{2_1}	a_{3_0}
27	a_{1_1}	a_{2_1}	a_{3_2}
13	a_{1_0}	a_{2_2}	a_{3_0}
*** 26**	a_{1_1}	a_{2_1}	a_{3_1}

of redundant tests. Table 1.5 illustrates a test suite based on random selection of $M = 18$ 3-way test cases from the exhaustive suite without replacement. This test suite hits one of the failing test cases, exhaustive number #26, on the 18^{th} trial.

$$1 - \left(1 - \frac{1}{18}\right)^M > \frac{1}{2} \tag{1.4}$$

$$M_{min} = \left\lfloor \frac{\log\left(1 - \frac{1}{2}\right)}{\log\left(1 - \frac{1}{18}\right)} + 1 \right\rfloor \tag{1.5}$$

1.6 Testing Network-Centric Software

As software grows in complexity - which in part is due to more network-centric communications - the importance of good software testing grows as well [38, 39, 60, 77]. Unfortunately, some aspects, such as security, seem to have received inadequate attention in practice resulting in software vulnerable to network-based attacks [78, 64]. In this work, the term "network-based software" will be used to denote software which uses some form of network-based communication

Table 1.5: Random Combinatorial Test Suite Example

Test Index	$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
5	a_{1_0}	a_{2_0}	a_{3_4}
31	a_{1_1}	a_{2_2}	a_{3_0}
27	a_{1_1}	a_{2_1}	a_{3_2}
9	a_{1_0}	a_{2_1}	a_{3_2}
17	a_{1_0}	a_{2_2}	a_{3_4}
18	a_{1_0}	a_{2_2}	a_{3_5}
15	a_{1_0}	a_{2_2}	a_{3_2}
34	a_{1_1}	a_{2_2}	a_{3_3}
6	a_{1_0}	a_{2_0}	a_{3_5}
35	a_{1_1}	a_{2_2}	a_{3_4}
1	a_{1_0}	a_{2_0}	a_{3_0}
19	a_{1_1}	a_{2_0}	a_{3_0}
8	a_{1_0}	a_{2_1}	a_{3_1}
14	a_{1_0}	a_{2_2}	a_{3_1}
30	a_{1_1}	a_{2_1}	a_{3_5}
10	a_{1_0}	a_{2_1}	a_{3_3}
32	a_{1_1}	a_{2_2}	a_{3_1}
29	a_{1_1}	a_{2_1}	a_{3_4}

during its operation.

There are many factors that play a role in the way software is tested. Resource constraints are often the primary driver (deadlines, personnel, cost, etc.). For instance, it is often impractical to test every permutation of all possible inputs (also known as a combinatorial exhaustive test). Unfortunately, a potential problem with selecting a sub-group of exhaustive tests is that a high risk defect could go undetected. Research shows that important failures are frequently the result of unexpected and unusual combinations and interactions (often correlated over time) of many parameters, faults and error-states [84, 4]. Therefore, it is important to investigate techniques that can help expose the underlying faults and at the same time adhere to a set of resource constraints [82].

Another important issue is the experience of testers or the tester expertise. While many current testing techniques may be effective when used properly, it has been shown that many testers lack the experience/expertise required to use them effectively [64]. Still other tools may be difficult to learn or use and may not become fully integrated into a testing process. There is a need for techniques that both increase the ability to test for unusual combinations and at the same time compensate for possible tester inexperience to help increase the overall effectiveness of testing [25].

Increasing effectiveness of testing requires increased defect exposure within specified

constraints. Pairwise testing [18, 75], enhanced with partial N-wise testing [7], may offer a viable option. The first step in making a case for the use of N-wise enhanced testing in the domain of unusual and unexpected failures and defects (such as security failures [77]), is to compare its effectiveness in that space with a technique such as random testing. Random testing is often used since it can be simple to implement and can be quite effective [4, 84, 12]. However, it has limitations [84]. For example, reliability analyses may be weak unless statistical coverage is guaranteed [5, 21]. The work reported in this chapter compares pairwise testing and random testing in the domain of network-centric software.

1.7 Motivation for Current Work

Random testing alone may not adequately test a system for faults, although it can be effective in some instances [50]. It is commonly used as an "add-on" technique to enhance a systematic process [21, 50, 47, 22, 61]. In fact, in order to adequately test software, a systematic process must be in place [54, 21, 50, 47, 22]. Schroeder et. al. define the set of all possible combinations of test values as T_{cart} which can be seen in Equation 1.6 [68]. This is basically the Cartesian product of all possible values of each sub-domain. The size of T_{cart} can be seen in Equation 1.7. On the other hand, systematic processes can be costly in terms of time and available resources and they may be certain types of faults that may be missed using a particular systematic process alone [21, 50].

In a highly complex environment, such as network-based system, it may be too costly not to test the lower probability sub-domains. It may be beneficial to adopt a process which combines random and systematic testing in a way that takes advantage of the efficiency of random testing and the effectiveness of some type of systematic testing for the low probability sub-domains. Unfortunately, there does not seem to be a consensus on how much random testing to perform vs. how much systematic testing to perform. Ntafos has suggested that both random and partition testing detect faults at about the same rate, except in the cases of "...low probability sub-domains that have high failure rates." [61]

Instead of picking a systematic technique over a random technique, it may be profitable to find a way to combine both. Utilizing combinatorial techniques may provide a viable systematic process to do just that.

Pairwise testing, perhaps complemented with partial or full N-wise testing, is a technique which guarantees that all important parametric N-tuples are included in a test suite. It is conjectured that N-wise enhanced pairwise testing can increase effectiveness of random testing in exposing

unusual or unexpected behaviors, such as security flaws. This testing could be also quite cost efficient since test suites grow linearly with the number of parameters when N is relatively small.

$$T_{cart} = \{D(x_1) \times D(x_2) \times \cdots \times D(x_N)\} \tag{1.6}$$

$$|T_{cart}| = |D(x_1)| \times |D(x_2)| \times \cdots \times |D(x_N)| \tag{1.7}$$

Some important questions in that context are:

1. Is there a way to effectively and efficiently combine random testing and systematic testing?
2. Is there a way to determine the amount of effort required to perform a systematic process if random testing is performed first?
3. Is there a way to determine the amount of effort required to perform random testing if a systematic process is performed first?
4. Is there a way to effectively and efficiently combine various types of systematic testing into one process?
5. Is it worthwhile to use statistical analysis from a systematic profile to perform this type of testing?

1.8 Dissertation Outline

This dissertation is composed of the following. Chapter 2 provides a closer look at how systematic testing can be used in a complex environments such as a networking environment. Chapter 3 discusses a hybrid method where N-wise testing is combined with random testing. Chapter 4 discusses how to optimize the hybrid approach and provides analysis for fault-detection properties of the proposed approach, and Chapter 5 provides summary and conclusion. Appendices provide further details.

Chapter 2

Testing in Complex Environments

Modern networking environments add a new and very complex component to most modern software-based systems. Interfacing over networks not only makes verification and validation of such software more complex, but it also makes such systems vulnerable to network-borne security attacks if one of the residual faults is exploitable via a network. Network-related security failures, but also other problems, can result from unusual and unexpected relationships among software/system parameters - input, output, internal and operational environmental parameters. Designer and tester expertise alone may not be sufficient to expose such exploits. This chapter investigates the issue of how to test for both normal and unusual relationships among software parameters in a systematic fashion, especially those related to complex relationships, such as network-associated operations. It also looks into problem-detection effectiveness of the approach. In this context, it explores methodologies, fault-detection, and cost effectiveness of different variants of a technique called N-wise testing.

2.1 Using N-wise testing

Pairwise testing is a specification-based testing strategy which requires, in principle, that every combination of valid values of any two input parameters of a system be covered by at least one test case. An extended variant of that is where every 3-tuple, 4-tuple, and in general N-tuple of the parametric values is covered. Empirical results show that pairwise testing is practical and effective for various types of software systems [18, 19, 17, 75]. This is supported by a recent study by Wallace and Kuhn [83] who analyzed 15 years of failure data from recalled medical devices. They conclude that 98% of the failures could have been detected in testing if all pairs of parameters had been tested

(i.e. 2-way testing) [68]. Of course, it is possible that higher-order (N-tuple) interactions, e.g., 5-tuple interactions, are needed to discover a fault and that such an interaction may appear by accident in a pair generated test-suite. If certain fault-types are known to require a high-order interaction for detection, it may be advisable to guarantee coverage of more N-way constructs for an $N > 2$.

Cohen and his colleagues were able to implement N-wise testing techniques on practical systems using heuristics [17]. Dalal and Cohen analyzed same-sized random test data sets and found them to cover a very high percentage of input pairs; as high as 90% in many cases [18, 19, 17]. Schroeder et al. define fault detection effectiveness (FDE) as the percentage of faults detected by a testing technique when executed on a system with known fault content [68]. Based on these hypothetical cases, one may not expect the FDE of pairwise and the random test data sets of the same size to vary a great deal.

Bolaki recently compared pairwise (and N-wise) testing techniques against random testing techniques [12]. He performed studies on two software systems that have been used in practice. One is called the Data Management Analysis System (DMAS), which is utilized by analytical chemists for data processing. The second is called the Loan Arranger System (LAS), which is used as a support system for mortgage-backed security businesses. He parameterized the inputs for both systems. More details regarding Bolaki's work can be found in Appendix E.) Of interest is what we call the defect degree, i.e., is the defect an 3-way, 5-way, and in general a k-way defect. That is, what is the minimum number of interacting parameters required to detect a particular fault or defect. This information is important since it tells us what level of N-way testing is needed to achieve a certain comfort level with respect to adequacy of the testing. Some of that information was not available in Bolaki's work, so it was derived. This is discussed further in one of the later subsections. Once Bolaki parameterized the systems, he used fault seeding to create faulty versions of the correct software one line at a time based on a particular fault model. According to Bolaki, only "simple" faults were injected into each system respectively. Bolaki then used an automated tool called the "Test Vector Generator" to construct the test suites based on particular criteria such as N-way relations, random, etc. He found that when the number of values per defined parameter is between 2 and 3 (on the average), random testing performs just as well as N-wise testing.

The situation is different in more complex systems and is different for other types of faults. The study described further in Section 2.2 has shown that, in a large number of instances, detection of security related faults required three or more interacting parameters. It was empirically determined that parameters involved in security failures may require about 3.3 interacting parameter values on average to detect a defect. Twenty-seven percent (27%) of the time the minimum number

of interacting parameters required to detect a defect was greater than 3, and the maximum number of values associated with one parameter was 10. The maximum number of parameters required to expose a defect was 4, which occurred about $\approx 10\%$ of the time. Because of suite size limitations, some of these configurations may never be exercised using random testing techniques.

On the other hand, and this is not surprising, there are indications that in practice N-wise testing has limitations. In testing Remote Agent Experiment (RAX) software used to control NASA spacecraft, Smith, et al. [73, 68] present their experience with N-wise testing. Their analysis indicates that pairwise testing detected 88% of the faults classified as correctness and convergence faults. Depending on the safety and cost criticality, this percentage may or may not be acceptable. Only 50% of the interface and engine faults were covered, which would be considered poor coverage for this type of system. Higher orders of N-wise testing could possibly find more faults where $2 < k \leq N$.

2.2 Assessing N-Wise Approach for Security Testing

Security issues and faults related to those issues are a subset of all other faults/flaws. However, security-related faults usually carry a higher risk factor and often arise as a result of potentially unforeseen dependencies, relationships, interactions, and constraints among the known system parameters [77]. The driving question is whether one can devise a formalized, systematic testing approach that allows automatic generation of test cases which cover 'unexpected' parameter interactions. Also of interest is the cost-effectiveness of the approach in terms of the number of test cases and in terms of finding security flaws.

N-wise testing offers a promising approach to this problem. It is important to determine the level of N-wise testing necessary given that $N > 2$ testing may considerably increase the complexity of the process and the number of test cases. The useful value of N can be determined a number of ways. One way is presented in the chapters that follow. All N-wise testing methods start out by assuming that all parameter groupings are independent of each other. Heuristics are then used to generate a suite of test cases that cover the N-tuple space. [1] These tests are then modified by explicit specification of inter-parameter dependencies and constraints [19, 75]. This tends to increase the number of test cases to provide additional coverage required by the relationships (dis-

[1] The In-Parameter-Order method [49] was used as the N-wise testing heuristic throughout this dissertation. An updated tool for PairTest v1.1 [14] was used to help automate part of the testing process for N-wise testing in general. To further facilitate experiments, and field use of findings, the PairTest tool was extended to include relations for N-wise testing.

cussed later in this chapter). Still, the number of test cases generated that way is far smaller than when all possible combinations of parameters are tested. This approach is also more repeatable and more likely to be automated than ad hoc approaches based on tester expertise. However, one would like to know the level of expertise that is required to produce appropriate N-wise relationships and constraints that provide similar or better fault (or flaw) coverage when compared to good ad hoc testing, good random testing, or testing using some other systematic approach. This is especially interesting in environments where 'unexpected' operational profile changes and bursts are the order of the day. An example of such an environment could be one which employs a network and depends on message communication. Unexpected changes and bursts could be the result of a number of things, such as malicious security attacks.

2.2.1 Data

The general approach taken in this part of the study is the following one. A set of known security flaws/problems/faults were collected from a public data repository at the Secure Space site [28]. [2] Those flaws were then parameterized, i.e., the conditions and inputs needed to describe the flaws were broken down and turned into failures based on individual parameters and associated parameter values. A set of parameterizations is show in Appendix D. Next those parameters were used to generate test cases using different parameter-based testing approaches - specifically N-wise method, random method, and expertise method. Finally, both the fault finding ability (effectiveness) of the different methods and the effort, number of test cases, and the algorithmic complexity (efficiency) needed to find the faults were compared.

The collection of known security flaws reported on-line for network-based devices [28] consisted of two sets of data. One set of 52 security flaws was reported in the database for switching devices from October 26, 2000 to June 9, 2004. Appendix D lists the 52 problems along side the risk level, flaw ID number, date of recording, and the prarmeterization applied to it. A more detailed description of each individual fault can be found at the Secure Space web site.

We used 47 of 52 faults to form a data set we call CORPORATE1. We also formed another set, which we call CORPORATE2, again using data from the Secure Space web site, that contained 23 flaws pertaining to a firewall category [3]. The published flaws were parameterized in the flaw space, and not in the platform space, i.e., the platform details were in most instances similar and were ignored to reduce the number of relatively-invariant parameter values. Then we used

[2] http://www.securityspace.com/smysecure/catdescr.html?cat=CISCO
[3] http://www.securityspace.com/smysecure/catdescr.html?cat=Firewalls

a tool called PairTest (Version 1.1) [14, 75] to generate test cases with different relationship and constraining strategies. For example, in one part of the study (described further in a section that follows) tester-driven pairwise test cases were developed for 55 parameters from CORPORATE1. Also, random relationships (mutual exclusivity of values and correlation of parameters) were automatically generated using our toolset. In this particular case, random relations were generated for pairs of parameters. Any parameters not explicitly included in the pairwise relations were guaranteed one-way coverage only. This means that every parametric value was guaranteed to appear at least once.

2.3 Detecting Security Faults

2.3.1 Transitivity

Security flaws are often exploited as a result of factors and relationships unknown at the time software is developed[77]. Therefore, it may not be sufficient to base security-related software testing and relational pairing rules on tester expertise alone. It may be worthwhile to investigate the effectiveness of an approach where relationships are randomly generated assuming pairwise [75], or N-wise in general, dependencies among different groupings of application parameters.

One goal was to determine the effectiveness of disclosing known security flaws using N-wise testing when dependencies amongst specified parameters are randomly chosen. Another goal was to determine the effectiveness of disclosing security flaws outside the space of known security flaws, i.e., to define parameters on one set of flaws and measure how effective that test suite is on another set of security flaws.

The latter resulted in an interesting observation on transitivity of effectiveness (ability to find faults) of pairwise testing under different strategies for generation of relationship rules and constraints. Pairwise test cases and relationships were generated using the extended version of the PairTest tool.

Figure 2.1 shows some of the results of the experiments. The vertical axis shows the number of faults remaining and the horizontal axis shows the number of test cases executed in increasing order. The top curve, CAT2 refers to application of the same pairwise test-suite to the CORPORATE2 category of flaws, the middle curve (CAT1) refers to the CORPORATE1 category of flaws, and there is an additional point where CORPORATE1 matches what was done based on random testing. Each point represents effectiveness (expressed as the number of flaws remaining)

of the total number of test cases generated using the CORPORATE1 flaw-set as the basis. The first point is for 9 test cases generated assuming total independence of CORPORATE1 parameters. This one detected only about 30% of the flaws. The second and third points are for test cases generated based on 10 and 20 random relationships, respectively. The fourth point is 38 test cases based on 30 general relationships created through human (tester) expertise rather than random generation of relationships. Detection rate increased to about 70%. Final point on CAT1 curve is the one where relationships were generated through ”full expertise” of a human tester. “Full expertise” refers to using only the tester's knowledge to formulate a test suite, which is a form of ad hoc testing. The application of “full expertise” of the author (64 relationships, 94 test cases) for this particular study increased the detection rate to about 90% or better.

A comparison of CORPORATE1 effectiveness of pairwise testing using 38 test cases and a random approach with the same number of test cases, shows that pairwise testing appears to be as effective as random testing with replacement. This confirms a recent finding by Bolaki [12]. This is good since the number of test cases needed to find problems with pairwise testing appears to grow slower than with random testing. It would also appear that application of expertise to the generation of these 38 random test cases may do as well (about 70% detection) as application of that expertise to produce 30 general relationships in the pairwise approach, so random testing with replacement seems to provide a lower bound for the effectiveness of N-wise testing.

Inspection of the relationships needed to detect flaws in the case of the CORPORATE1 data set indicates that, on average, the minimum number of parameters required to interact in order to detect a known security flaw was at least 3. This suggests that it may be necessary to routinely generate at least 3-way relations (and perhaps some of higher order) to adequately detect some security flaws.

When the same test cases were applied to a different security domain (in this case firewall related - CAT2 designation in the figure), the number of remaining defaults reached a plateau at about 80%. Beyond that, domain-specific parameters needed to be introduced and additional expertise needed to be applied. This result indicates that, as might be expected, that systematic strategies cannot be applied automatically. Careful thought has to go into design of the testing conditions and test cases, or the effectiveness may be less than acceptable even when dealing with “same” impact domains (in this case security faults).

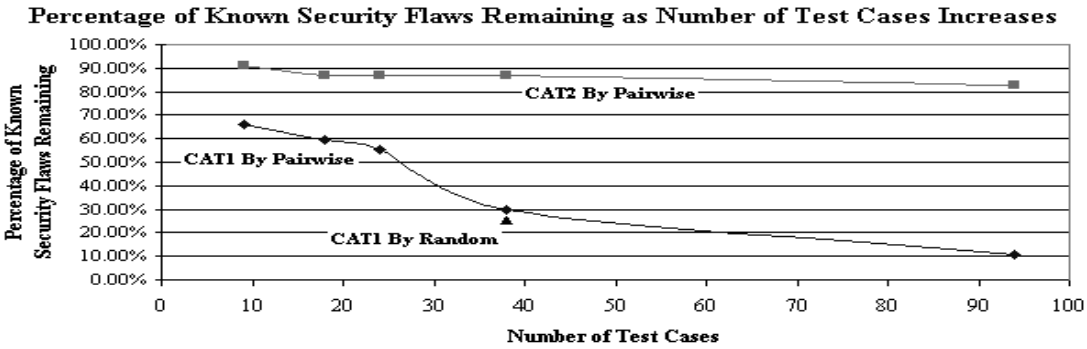

Figure 2.1: Effectiveness of Pairwise Testing in a Security Setting

2.3.2 Expertise

Many current testing techniques may be effective when used properly. However, it has been shown that many testers lack the experience required to use them effectively [64]. Also, tools are sometimes difficult to learn and may not become fully integrated into a testing process. There is a need for techniques that both increase the ability to test for unusual combinations and at the same time compensate for possible tester inexperience to help increase the overall effectiveness of testing [25]. What is the role of expertise in the pairwise context?

Pairwise test cases were generated by progressively increasing the number of specific relationships and constraints. One has to distinguish between full N-way testing (generation of all N-tuples for a given full set of parameters) and 2-parameter testing, where all pairs are generated only for a subset (in this case in pairs) of the parameters, the rest being chosen randomly. When only pairs of parameters are used, then the relationship assumed between only the two chosen parameters and all pairs of values two particular parameters cover is guaranteed. The rest of the pairs may or may not be present, but all values are guaranteed to be there at least once.

When running test cases, relationships found in the test-cases are compared with combinations of the parametric values in each flaw-case. For the CORPORATE1 set the interacting parameters are listed in the last column of Appendix D while the actual full set of parameters are listed in Appendix C. If there was a match between relationships in the test-case and relationships in the flaw/fault table, it was assumed that the fault was detected.

One also has to remember that although pairwise relationships are generated deliberately and pair-coverage is the goal, 3-way, 4-way and higher order relationship are also generated as part of each multi-parameter test case. A related issue is how the relationships are generated, e.g.,

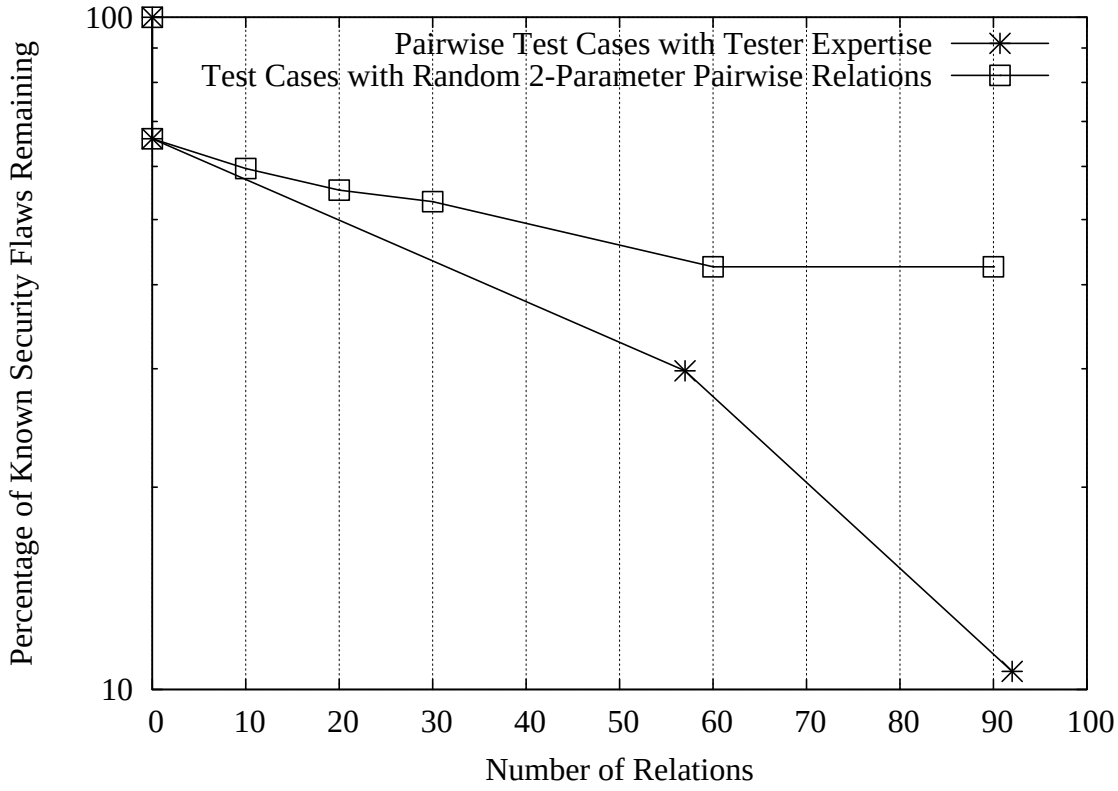

Figure 2.2: The Effectiveness of Pairwise Tests Based on Random Parametric Pairs (Relations) and Pairwise Tests with Relations Based on Tester Expertise

randomly or by hand, and the role of human expertise.

Figure 2.2 explores the issue further. The vertical axis is logarithmic and shows the percentage of known flaws remaining (CORPORATE1 data set). The horizontal axis is the number of relationships generated in the tests. The first thing to note is that random generation of relations is less effective in finding problems than an expertise-driven test suite with a corresponding number of relations. This is not unexpected. However, the random approach has lower overhead than the expert-only approach.

In the case of the random relations, it took an initial overhead of approximately 3 days for a program to automatically generate a number of relations and generate a set of test cases, accordingly. Then from that point on, it was a matter of 1 to 2 minutes to generate test cases with different random sets of 2-parameter pairwise relations. When this is extended to include pairwise relations amongst a greater number of parameters, one would expect a longer generation time as well as an increase in the number of test cases. On the other hand, it took approximately 7 days to

manually generate 57 general relations based on one researcher's expertise. It also took another 4 hours to extend the number of relations to 92. Using the randomly generated relations for pairwise testing could have resulted in over 300,000 automated test executions in the same amount of time it took to generate the relations by hand!

2.3.3 Random

There appears to be a definite advantage in using random relations and random testing for long runs, but it does take longer to find all the problems. If the cost of failure identification and fault repair is high, using fewer test cases may be a better option. On the other hand, the number of test cases generated by the random 2-parameter pairwise-relations process seems to increase at about the same rate as the number of test cases generated manually, but it would appear that the expert approach takes a somewhat straighter "path". This can be seen in Figure 2.3. The figure shows the number of test cases generated for CORPORATE1 fault-suite versus the corresponding number of relations.

The expert approach and generation of random relations appear to have similar effectiveness, at least in the range examined here. This is shown in Figure 2.4, which plots the effectiveness of testing versus the number of test cases generated. The stopping criterion for the randomly generated tests in this figure was the number of pairwise test cases generated.

It would appear that a very simple approach to randomly choosing relations may be less than favorable. Since high-order relations are also chosen somewhat randomly when a particular full k-way testing is applied in a p parameter space, it may be more advantageous to follow up k-way testing with random testing in order to cover additional higher-order relationships. An open question is whether guaranteeing pairwise testing for more than 2 parameters at a time and larger depth of parameter values makes the random relations approach increasingly more effective. Automated N-wise testing may then be useful when combined with an existing testing technique with the intent of disclosing new security faults.

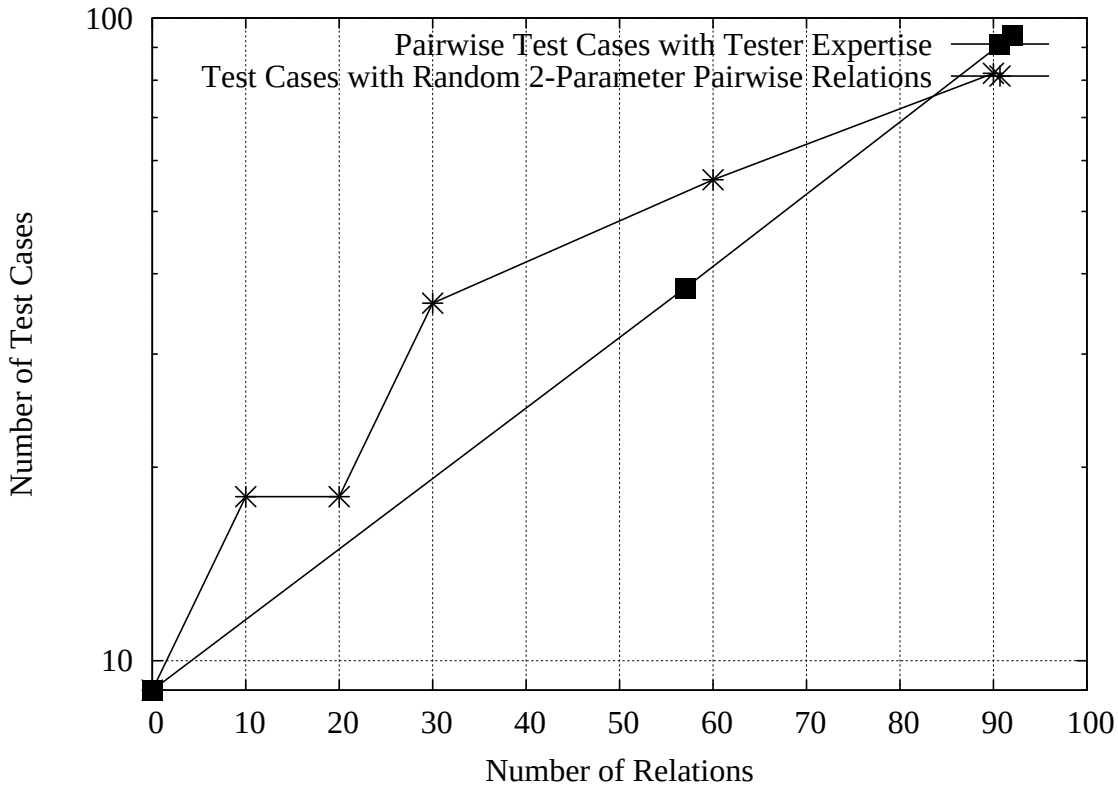

Figure 2.3: Comparison of CORPORATE1 Rate of Increase for the Number of Pairwise Tests as the Number of Relations Increases

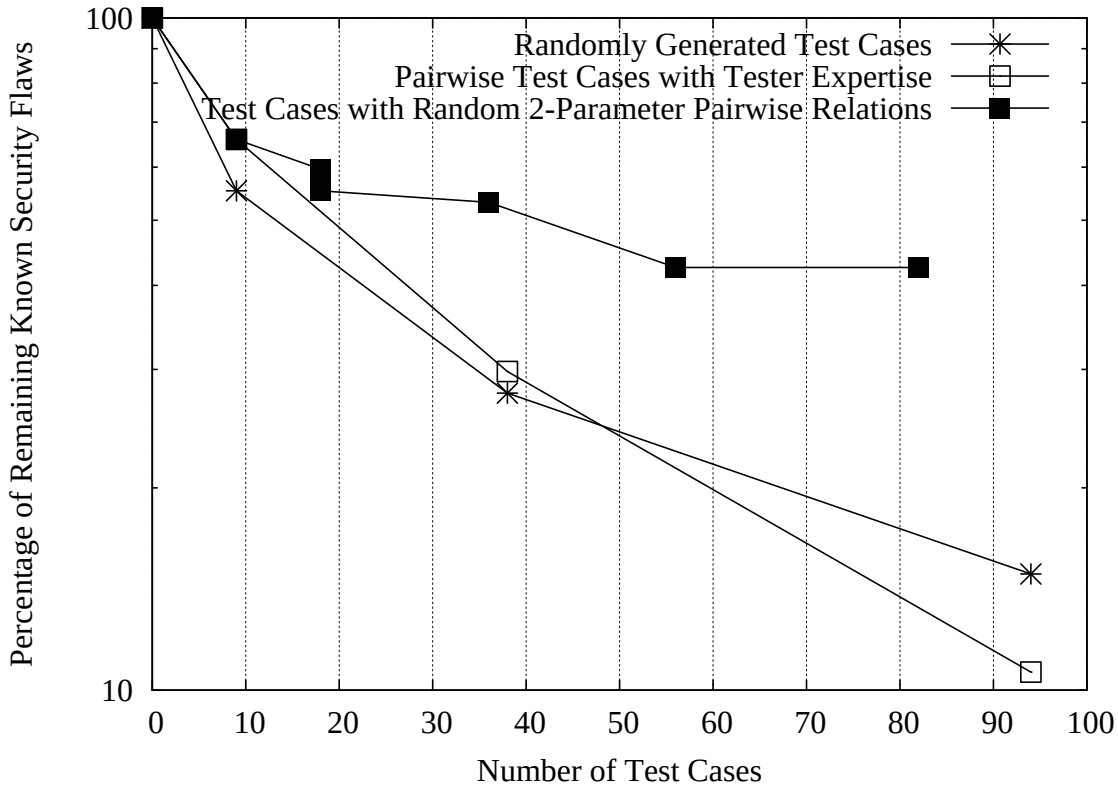

Figure 2.4: Comparison of CORPORATE1 Number of Remaining Known Security Flaws as the Number of Test Cases Increases

Chapter 3

Hybrid Software Testing

Hybrid approaches combine one or more techniques. This chapter focuses on the efficiency and effectiveness of a testing approach that combines N-wise testing with random testing. Only discrete input spaces and faults - called N-way faults - are considered (i.e. faults detectable by an N-way combination of system/software input parameter values). An implied assumption is that full N-wise testing may be too expensive when the number of parameters and parameter values is large, and that some testing resources should be devoted to some level of N-wise testing without replacement, while the remaining resources be devoted resources to random testing with replacement.

The cost of generating random operational profile conformant test cases is typically $O(M \times p)$ where p is the number of parameters and M is the number of test cases. (See Algorithm RandomWreplace described in Section 4.4.1.) Unfortunately, random testing can result in large test suites, may take long to execute, and results may take long to analyze if an oracle is unavailable. Systematic approaches can generate smaller test suites, reducing run-time and analysis costs, but can take much longer to generate since they can require a higher level of initial expertise to develop. We must question effectiveness versus the efficiency with both random and systematic testing approaches.

Stopping criteria that maximize testing efficiency (i.e., minimize costs) while maximizing effectiveness (i.e., number of detected problems) is an ongoing theme [4, 12, 24, 23, 32, 52, 56, 55]. In this context, statistical (usually random) approaches, systematic approaches, and ad hoc approaches are assumed to be distinguishable. Statistical and systematic approaches often have more realistic measures associated with them that can be used to define stopping criteria.

A problem with statistical methods (which include random testing) is that they may be

efficient (or at least may appear to be efficient), but they may not be effective. A problem with systematic methods is that they may be effective (or at least may appear to be effective) for certain types of faults, but are inefficient because of the extra effort and specialized knowledge that may be needed to set-up testing and generate test cases. Ad hoc methods sometimes yield excellent results, but may not be repeatable, thus they tend to escape systematic quantitative analysis.

Efficiency can be measured through effort (e.g., person-hours) and test suite time or size. The ability to remove faults or failures is a measure of effectiveness. [1] In general, efficiency is more than just the cost of test generation. However, a failure needs to be identified (which requires computer and personnel time to examine results), and then a failure needs to be corrected. The costs of the failure identification and correction process may cover a large range. According to Musa et al. [57] (F5.11-13, p.138-139) the costs of failure identification can include the following:

1. 3 - 10 person-hours per CPU hour of testing (varies with machine speed),
2. 0.2 - 2 person-hours for each failure identified, and
3. , x/2 CPU units to confirm and report each failure identified (given computer time of x CPU units to find a failure)

For failure correction one may need 1 - 5 person-hours to identify fault, and another 1 - 5 person-hours to correct the fault. In addition to the 2-10 hours of person and computer time is the cost of test case development and generation. Of course, the values may and do differ with different applications and methods for determining result correctness.

In this work, the focus is the test case generation component of the costs. We assume that the failure identification and fault correction costs are comparable for all methods considered.

There also are many ways to quantify effectiveness (e.g., [16, 15]). Recall that a measure of effectiveness is that ability of an approach to remove faults or failures. More specifically, effectiveness measure could be "...the probability that at least one failure is detected with a specified sequence of test cases [15]." Another measure could be the number of distinct test cases needed before the first fault is detected. A third possible measure could be fault coverage. The number of faults found given a fixed number of test cases is yet another measure. For example, given that M test cases are generated, one may want to get the largest possible return in terms of the number of

[1] Human or other errors result in insertion of faults/defects into a software artifact. Those faults and defects may propagate through the software process into the executable product. During execution when a fault/defect is encountered, a software system may go into an error-state. That error-state alone, or in combination with other error-states and environmental conditions, may result in an externally observable behavioral anomaly. Such an event is called a failure.

faults found. Another possibility is the following: given that one may want to find X faults, one may wish to determine the number of test cases that would be needed. In the next section, efficiency and effectiveness are defined as they are considered in this chapter.

3.1 Developing a Hybrid

The hybrid model of interest is the one where N-wise testing is followed by random testing. Since we are targeting N-way faults, it would more straightforward to just use N-way testing. Unfortunately, it may be difficult to guess what the largest number of interacting parameters that results in a failure may be in practice. One could estimate the number from historical records (e.g., requirements errors and faults) for the system under development, or from information on similar systems elsewhere (e.g., security faults discussed in Chapter 2). Therefore, practical considerations may stop us from generating more than a certain level of N-wise testing. In this context, consider the following.

Let p be the number of input parameters. Let each parameter take on a number of discrete values (v_i such that $i = 1 \ldots p$). Then an input (or test case) vector is p values long and its elements are the allowed parameter values (one from each parameter). Selection of parameter values is a matter of the testing approach.

In pairwise testing, all pairs of parameter values are represented in a pairwise complete test suite. Hence, pairwise testing will be completely effective for all 2-way faults that can be discovered by exhaustive coverage of all discrete input parameter pair values.

Pairwise test suites are usually much smaller than corresponding random test suites that achieve the same fault coverage or the same effectiveness [19]. Note that when the number of parameters is $p > 2$ a test vector will contain tuples of higher order. Hence, pairwise testing guarantees coverage of all 2-tuples, a proportion of all 3-tuples, and up to a proportion of all p-tuples. Obviously, if a system contains faults that can be detected only by a k-tuple where $k > 2$, then pairwise testing may miss some of those faults unless coverage of the k-tuple space is complete. Therefore, we ask: If k-wise testing catches a fraction of higher-level N-way faults, can that fraction be increased through random sampling? This question assumes that random sampling would be easier to affect than high-level N-way testing.

First, let us examine some effectiveness issues of random testing, and later return to the hybrid model..

3.1.1 Effectiveness of Random Testing

Let the input space of a system S be discrete and let it be describable by p parameters, where each parameter can have a certain number of discrete values. To simplify, let all parameters have the same number of values, v. Then exhaustive coverage of this domain requires v^p combinations of parameter values, i.e., all possible combinations of p parameter values. Let a test case for system S be a vector of p single input values from each parameter. Let the number of faults in S be m and enumerated from 1 to m. Then let t_i represent the tuple size for the i^{th} defect where $1 \leq i \leq m$.

The parameter space that needs to be covered by the test cases to guarantee detection of a t_i-tuple detectable fault has a size of at least v^{t_i}. Let the largest value be t = $max(t_i) : 1 \leq i \leq m$. t-way coverage of the parameter space should then detect all m faults. Let the value, k be a tuple size such that $2 \leq k \leq t$.

If it is assumed that faults are uniformly distributed over the parameter space, an estimate of the probability of detecting one of the m faults by a p-tuple test vector (test case) is $\theta = m/v^p$. However, a single test vector may contain one or more t-tuples if $t \leq p$. So a finer granularity view may be in order. The t-tuple space is v^t times the number of ways one can select a particular set of t parameter values. Let the number of ways in which one can choose t parameters out of p be denoted by $C(p, t)$. Then an estimate of the probability of detecting a t-tuple detectable fault is Equation 3.1.

$$\theta' = \frac{m}{C(p,t)v^t} \tag{3.1}$$

As mentioned before, one can define effectiveness of random testing in several ways. The key value in this analysis is the probability that a random sample of the parameter space discovers a fault. This probability can then be combined with the process and other assumptions in order to assess effectiveness in terms of the number of test cases needed to achieve the goal.

For example, one could consider a sequence of M random test cases and estimate the probability that they would find m failures (without removing the faults, i.e., sampling of the fault space with replacement). This would be described by the binomial probability mass function. If the faults were removed when found (i.e., sampling without replacement), an appropriate distribution could be the hypergeometric distribution. Another metric could be to count the number of test cases that might have to run until (and include) the first fault is found. The probability that M random test cases will run until one t-tuple sensitive fault is detected is then described by the geometric distribution [79], given in Equation 3.2. This measure is similar to the F-measure discussed by

Chen et al. [15]. On the other hand, if it is desired that the probability that M random test cases are needed before the first fault is detected, it can be given by the modified geometric distribution [79].

$$P(F = M) = (1 - \theta')^{M-1} \theta' \tag{3.2}$$

3.1.2 When is random testing adequate?

One way to pose the question of how to combine systematic and random testing is to ask "When random testing perhaps may not need enhancement in dealing with a system which has parallel discrete inputs (i.e., p parallel input values, or a p-long vector of input values)?" As before, we assume that defects are evenly distributed across combinations of t of the p parameters, i.e. over C(p,t) t-tuple slots. This assumption means that one grouping of t parameters has approximately $\frac{m}{C(p,t)}$ defects associated with it. Since θ' is a probability, $0 < \theta' < 1$. [2] From Equation 3.1 then the following can be deduced: $\frac{m}{C(p,t)} < v^t$ or $m < (v^t \times C(p,t))$.

The θ' described in Equation 3.1 considers the special case of when all the parameters have the same domain size, i.e. all parameters have the same number of values. Now consider the general case where the number of values associated with each parameter can vary. Then instead of v^t which occurs in the special case, each tuple of t parameters would be a multiple of t varying numbers of values which is represented by Equation 3.3.

$$\Pi_{j=1}^{t} v_{i,j} \tag{3.3}$$

Let $v_{i,j}$ represent the j^{th} parameter in the i^{th} grouping of t parameters where $1 \leq j \leq t$ and $1 \leq i \leq C(p,t)$ since there are C(p,t) possible groupings of t parameters. For example, suppose there are 3 parameters - P_1, P_2, and P_3 - that have a total of 2, 3, and 4 values associated with each respectively. Now suppose t=2, which means all possible pairs among these three parameters are desired. Parameter pairs are as follows: $\{\{P_1, P_2\}, \{P_1, P_3\}, \{P_2, P_3\}\}$, which is a total of $C(3,2) = 3$ pairs. Suppose the pairs are enumerated sequentially from left to right where the first pair is enumerated as 1. Let the parameters within each tuple likewise be enumerated. Then $v_{3,2}$ would represent a value from the second parameter from the third pair(i.e. a value from P_3 from the $\{P_2, P_3\}$ pair).

[2] It is assumed that θ' is greater than 0, else it would generally mean that no technique - whether random or systematic - would be able to detect the defect. Likewise θ' is assumed to never be 1 since that would mean that the defect would be detected in every situation regardless of the technique chosen.

Now let P_j represent the j^{th} parameter in a particular grouping of t parameters where $1 \leq j \leq t$, which is a t-tuple of parameters. There are a total of C(p,t) possible groupings of t parameters at a time in one test case across all p parameters. Suppose that each grouping of t parameters has a particular probability of detecting a defect associated with it in one test case. Let that probability be denoted θ_r where r is the r^{th} grouping of t parameters. Since there are a total of C(p,t) possible groupings, then $1 \leq r \leq C(p,t)$. Since all parameters are tested simultaneously, there is some dependence upon the probability of detecting a defect amongst all parameters in a single test case, i.e., there may be some overlap. In general, the union of these probabilities takes this overlap into consideration so that the general overall probability of detecting a defect in a single test case can be presented by Equation 3.4.

$$\theta = \{\theta'_1 \bigcup \theta'_2 \bigcup \cdots \bigcup \theta'_{C(p,t)}\} \tag{3.4}$$

When $\theta'_i = \theta'_j | \forall \theta'_i, \theta'_j \in \theta, i \neq j$ then this is referred to as θ'. This occurs when $\Pi_{j=1}^{t} v_{1,j} = \Pi_{j=1}^{t} v_{2,j} = \cdots = \Pi_{j=1}^{t} v_{C(p,t),j}$ which is simply referred to as v^t in the special case when all parameters have the same number of values. Equation 3.5 shows the general situation for k-tuple testing where $2 \leq k \leq t$.

$$\theta'_i = \frac{m}{C(p,k)\,\Pi_{j=1}^{k} v_{i,j}} \tag{3.5}$$

Let's assume that random testing needs to be enhanced when the probability of finding at least one defect with M test cases (sampling without replacement) is 'less than' some value X. Let the probability of finding a fault on a single trial be θ'. Let X be 1/2 (i.e., there is a greater than even chance of finding at least one fault by running M test cases). Of course, an estimate of the θ' must be made independently. So, the probability of not finding a fault on one trial is $1 - \theta'$, the probability of not finding a fault in M trials is $(1-\theta')^M$, and the probability of finding at least one fault with M test cases is $1 - (1-\theta')^M$. Then, assuming independence and sampling with replacement and M trials, the unsatisfactory outcome is when the probability of finding at least one fault is less than 1/2, as can be seen in Equation 3.6. Substituting for θ' and solving for v in Equations 3.7 to 3.15 shows that once v exceeds a certain value, random testing may not yield satisfactory results for a given m, t, p and M under these adequacy assumptions. Of course, this example can be recalculated using a smaller value than 1/2 to bracket the acceptable size of v. What the relationship says is that if, on average, there is a large number of parameter values, the probability of randomly producing a test

case that has a t-way tuple that detects a fault may be less than the acceptable adequacy criterion. At that point one could decide that it may be more effective to supplement random testing with a method that has a higher probability of finding a fault.

A similar argument can be made when considering k-wise testing. While k-wise testing assures coverage of all k-tuples, it also generates some higher-order tuples. If we assume that higher order tuples are generated relatively randomly (although randomness will be restricted due to the focus on k-tuple generation), then k-wise testing will behave somewhat like random testing where $t > k$ tuples are concerned and will have the same deficiency as random testing when it comes to detection of higher order N-way faults.

Equations 3.6 through 3.15 show that there is a break-even point which depends on the number of values a parameter has beyond which a method more focused on t-tuple sensitive faults may be needed to satisfy the adequacy criterion stated previously.

$$1 - (1 - \theta')^M < \frac{1}{2} \tag{3.6}$$

$$\frac{1}{2} < (1 - \theta')^M \tag{3.7}$$

$$\log_2(\frac{1}{2}) < M \log_2(1 - \theta') \tag{3.8}$$

$$\frac{\log_2(\frac{1}{2})}{M} < \log_2(1 - \theta') \tag{3.9}$$

$$\frac{-1}{M} < \log_2(1 - \theta') \tag{3.10}$$

$$2^{\frac{-1}{M}} < 1 - \theta' \tag{3.11}$$

$$\theta' < 1 - 2^{\frac{-1}{M}} \tag{3.12}$$

$$\frac{m}{C(p,t)\, v^t} < 1 - 2^{\frac{-1}{M}} \tag{3.13}$$

$$v^t > \frac{m}{C(p,t)} \times \frac{1}{1 - 2^{\frac{-1}{M}}} \tag{3.14}$$

$$v > \sqrt[t]{\frac{m}{C(p,t)} \times \frac{1}{1 - 2^{\frac{-1}{M}}}} \tag{3.15}$$

Let's look at the problem from a somewhat different perspective. Let's ask if there is an input space size (defined by a combination of p and v) in which random testing alone could be effective. One could use equation Equation 3.15, or one could strengthen the adequacy criterion and focus on the probability of detecting a single fault by requiring that value be greater than some

value X, say 1/2.

Equation 3.16 shows the situation after substituting in for θ'. The probability of a fault being found with uniform random testing (with replacement) is given by Equation 3.1 when all the sub-domain sizes are equivalent; and, this inequality can then be expressed in terms of v^t (Equation 3.17). The inequality can then be used to determine both the maximum and minimum number of values, represented as v_{max} and v_{min} respectively, in terms of tuple size (t), number of known defects (m) and number of parameters, p (Equations 3.18 and 3.19). Of course, if faults are removed these values will change during the testing process.

$$\frac{1}{2} < \frac{m}{C(p,t) \times v^t} < 1 \tag{3.16}$$

$$\frac{m}{C(p,t)} < v^t < \frac{2m}{C(p,t)} \tag{3.17}$$

$$v_{max} = \left\lceil \sqrt[t]{\frac{2m}{C(p,t)}} - 1 \right\rceil \tag{3.18}$$

$$v_{min} = \left\lfloor \sqrt[t]{\frac{m}{C(p,t)}} + 1 \right\rfloor \tag{3.19}$$

Table 3.1 gives a small illustration of when random testing may adequately detect defects alone. As before, p is the number of parameters, m is the number of known defects, M is the number of test cases, t is the maximum tuple size of the defects, C(p,t) is the total number of t-tuples among p parameters at a time. All these variables are used to calculate v_{max}, which is the maximum number of values in which random testing can be effective under the "1/2 first detect" criteria.

In the first row, $v_{max} = 170$. It means that the upper bound on the size of v is 170 values. So long as none of the parameters have more than that number of values, random testing has a better than 50:50 chance of finding a defect, and a very high probability of finding many of them with million test cases. In this case, random testing may not need enhancing for finding most (perhaps all) 2-way (pair) defects. In the second row, the maximum for v is 9. If a higher complexity system,may have more than 9 values in a parameter. then to finding 3-way defects effectively, this testing may need an enhancement. In rows 3 and 4, random testing component may be adequate for very small numbers of values and running a 4-wise or higher test suite before resorting to random testing may be advisable. Problems could occur once a tester begins looking for 4-way and 5-way

defects for parameters with 3 or more values. In row 5, one million test cases may not be sufficient to catch any 6-way defects. Random testing does not appear to be adequate since v_{max} is 0 in this case. This would mean that one would expect that testing needs to be enhanced by a $t = 6$ sensitive technique. Similarly the table can be used to illustrate that, given pairwise testing, its random component (which is really sampling without replacement) may start losing fault detection power very quickly when it comes to higher order tuples. At that point, it may be more effective to use full random testing (of p sized space) to supplement it.

The size of an N-wise test suite is discussed in the next chapter. It is worth noting, though that while the size can be quite manageable for t=2, it grows exponentially for large parameter and value spaces.

Table 3.1: Calculating the Maximum Number of Values Required to Use Random Testing Alone (v_{max})

p	m	M	t	C(p,t)	v_{max}
100	100	1e6	2	4950	170
100	100	1e6	3	161700	9
100	100	1e6	4	3921225	2
100	100	1e6	5	75287520	1
100	100	1e6	6	1192052400	0

3.2 A Hybrid of k-wise and Random Testing

Now consider the following. In the simplest case, a hybrid approach could be a combination of pairwise testing as the systematic testing approach. It minimally guarantees coverage of all defect pairs while keeping the test suite size relatively small. Test cases are generated using the pairwise approach via the In-Parameter-Order heuristic [49], which guarantees that all possible values of parameters grouped two at a time would be covered in the test suite.

One can estimate the portion of higher order t-tuples that can be covered using a pairwise suite ($k = 2$). The total pairwise space domain size for one grouping of a pair of parameters can be represented using v^2. This is a lower bound on the test suite required to cover all pairs and a portion of all t-tuples. The actual sizes will most likely be larger than the minimums noted based on the heuristic used to generate the test suite.

The minimal criteria could be used to estimate the proportion of the coverage of higher order tuples, which in turn could be used to estimate the fraction of the higher order t-tuple defects that may be caught using the lower order pairwise testing. This portion will be called β.

In the simplest case of pairwise testing, $k = 2$ and $\beta = \frac{1}{v^{(t-2)}}$. In the more general case, the test suite size fraction for tuples of size t using k-wise testing where $2 \leq k \leq t < p$ can be seen in Equation 3.20.

$$\beta \approx \frac{v^k}{v^t} = \frac{1}{v^{(t-k)}} \tag{3.20}$$

For example, if $v = 6$, $p = 100$ and there are 100 5-way defects, one may wish to gauge the following:

- What fraction of 5-way tuples is 4-wise testing guaranteed to cover?
- What is the estimated fraction of defects that 4-wise testing may detect?
- Given a set number of random test cases, how many new 5-way defects may random testing catch after 4-wise testing concludes?

Using Equation 3.20, one can estimate that about $\frac{6^4}{6^5}$ of all 5-tuples will be covered, which is $\frac{1}{6}$ of all 5-tuples. It can then be estimated that 4-wise testing may detect $\frac{1}{6} \times 100 = 16$ of the 100 5-way defects. There are a total of $6^5 \times C(100, 5) = 9e12$ 5-way constructs (which will be called CT). The probability of detecting a defect in one grouping of t parameters in one test case using random testing is $\frac{100}{CT} = 1.07e - 11$. Assuming that full 5-wise testing would require exactly $6^5 = 7776$ test cases. 4-wise testing requires exactly $6^4 = 1296$ test cases. Random testing would require an M based on Equation 3.22 which is at least 4057931819 test cases to catch at least one defect with more than a probability of 1/2 based on the geometric distribution from Equation 3.21.

Using a variation of the In-Parameter-Order (IPO) heuristic, 4-wise testing would produce 1844 test cases and 5-wise would produce 11070 test cases. This is based on an equation shown in the next chapter and can be seen in Section 4.5. Using the optimal number of 5-way tests as the total number of test for the 4-wise hybrid testing approach leaves $7776 - 1844 = 5932$ random tests that can be run. This is a savings of $1 - \frac{7776}{11070} \approx 30\%$ in suite size over conducting full 5-wise testing. It guarantees that $\frac{1}{6} \times 7776 = 1296$ of all 5-tuples are covered as well. There are assumed to be $100/C(100, 5) = 1.32e - 6$ defects associated with each grouping of 5 parameters. There may be overlap between the actual number of defects that can be caught using 4-wise and the ones using random. Some 5-tuples may be missed. Full 5-wise testing would of course catch all 5-way defects.

Figure 3.1 shows the effectiveness versus efficiency of the method. The number of remaining constructs has been normalized on the y-axis. The hybrid seemed to cover comparatively

as many as the full 5-way testing when the cutoff point was 10^5 test cases. There were approximately twice as many 5-way constructs covered by 5-wise testing than with 4-wise hybrid testing. This is an improvement over 4-wise testing alone which is expected to cover 1/10 the 5-way tuples covered by 5-wise testing. It is able to around cover 5 times the 5-way constructs covered by 4-wise testing alone. While in this example, the extra 6990 test cases may be justified for performing full 5-wise testing, it may become infeasible in more complex situations. Given the stopping criteria listed, the point at which the full 5-way testing seemed to begin to cover 5-way tuples faster than the 4-wise hybrid was right around 30000 test cases. This would suggest that the savings from performing full k-wise testing may not be noticed until much further into testing. Depending on the system, 30000 test cases to analyze at the beginning may prevent one from seeing the benefits of implementing full 5-wise testing over the 4-wise hybrid approach.

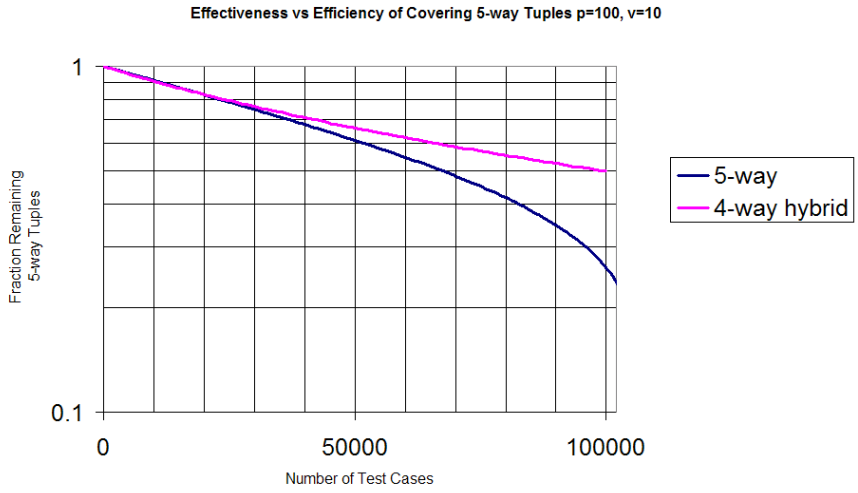

Figure 3.1: Effectiveness vs Efficiency Comparing Hybrid Testing to 5-way Testing to Cover 5-Way Tuples within 10^5 Test Cases

A minimum subset of random test cases were selected based on the minimum number of test cases needed to make the probability of finding at least one defect with probability greater than 1/2. The number of random test cases could be more, but the desire is to keep the test suite as small as possible, while being effective. The probability of finding at least one defect can be described using the geometric distribution as can be seen back in Equation 3.2 [79] where M is the number of trials (which in this case is equivalent to the number of test cases). However now, the geometric distribution will become an inequality since it is desired that the probability ($P(F = M)$)

in Equation 3.2 be greater than 1/2 which is represented in Equation 3.21. Suppose one criterion is to find the minimum M needed to make the inequality in Equation 3.21 true. The equation can then be manipulated to solve for M with the probability of detecting at least one defect set to 1/2, and θ' is the probability of finding a defect in one test case represented by Equation 3.1. The resulting minimum M can be seen in Equation 3.22.

$$1 - \left(1 - \frac{m}{C(p,t) \times v^t}\right)^M > \frac{1}{2} \tag{3.21}$$

$$M = \left\lfloor \frac{\log\left(\frac{1}{2}\right)}{\log\left(1 - \frac{m}{C(p,t) \times v^t}\right)} + 1 \right\rfloor \tag{3.22}$$

This example illustrates that a hybrid of pairwise systematic testing followed by random testing with replacement with minimal stopping criteria produced a process that was at least as efficient as random testing and was more effective than pairwise testing alone when the defects were at least 5-way.

Chapter 4

Optimizing the Hybrid Approach

4.1 Notation

C(r,u) = the combinatorial representation commonly referred to as "r choose u." [1]

p = the number of input parameters for a system.

t = the maximum number of interacting parameters in a defect.

k = the number of parameters that are related (or depend on each other). This implies that all combinations of k-tuples need to be covered by the testing. In general, $2 \leq k \leq t$.

m = the total number of known t-tuple defects in the system.

v = the number of values associated with each parameter. While the number of value may differ for each parameter in an actual system, for simplicity it will be assumed in this text that all parameters have the same number of values. In the examples, the average number of values over all parameters may be used as the value of v.

OPT = the optimal number of test cases to cover all t-tuples - v^t - which is the size of the space of t-tuples for one grouping of t parameters. This is a minimum bound for the actual size of a test suite which guarantees to cover all t-tuples across all parameters grouped t at a time.

AllSize = the size of the maximally exhaustive test suite (which includes all parameters), i.e., all combinations of all values of system parameters, typically grouped into p-long input vectors.

[1] This may also be seen as $\binom{r}{u}$.

$\mathbf{x}_{max}$ = the maximum number of random test cases needed to theoretically find all (known) defects in a software-based system with some probability X. In this text X is typically larger than 0.5.

4.2 Why Optimize?

One would like to optimize the level, number and the mix of N-wise test cases (typically sampling without replacement) and random test cases (sampling with replacement). Some optimization may also be needed to compensate for the additional test cases N-wise testing could produce since N-wise testing relies on heuristics. Thus, it is possible to have a (much) larger test suite than the optimal N-wise testing suite would be. The optimal number of test cases for t-wise testing level would be close to v^t. The number of extra test cases may vary depending on the heuristic in use. While N-wise testing guarantees that none of the test cases (p-tuples) are repeated, it does not guarantee that some of the t-tuples will not be repeated.

Since N-wise testing is based on a heuristic, it may take a considerable amount of time to generate the test suite, depending on the k-level desired and the number of values associated with the parameters. In the case of the In-Parameter-Order heuristic, Lei et al. found that the algorithmic complexity was $O(v^5 \times p^2)$ for pairwise testing (k=2) [49]. This can be rewritten as $O(v^{(2k+1)} \times p^k)$. It can be shown that this expression holds for general k-wise testing as long as the algorithm used to generate test-cases is similar to pairwise testing, which was the case with the extended PairTest tool used in this work As long as $k \ll p$, the algorithmic complexity will be a polynomial. However, if k became large enough, the complexity could potentially become $O(v^p \times p^p)$ which is then exponential and no longer desired.

In order to combat these issues, one optimizes by limiting the N-wise testing to some integer level k. If one wishes to use N-wise testing to cover all t-tuples, one could set $N = t$ and perform the testing unless t caused one of the aforementioned problems. In that case it would be preferable to choose a maximal k such that $2 \leq k < t$ and then possibly follow that with random testing. The premise behind picking the maximal k is that more unique t-tuples are guaranteed to be covered than with smaller k's. In fact, performing a maximal k-wise testing guarantees coverage of all smaller N-tuples, as illustrated in Figure 4.1. Performing k-wise testing subsumes all k-tuples, (k-1)-tuples, ..., and 2-tuples (or pairs), (see figure 4.2).

In figure 4.2, the coverage of all t-tuples is desired, but there is a maximum number of resources expressed as the number of test cases available. This maximum number could be based on the optimal number of test cases expected to cover all t-tuples, or on some other metric. The

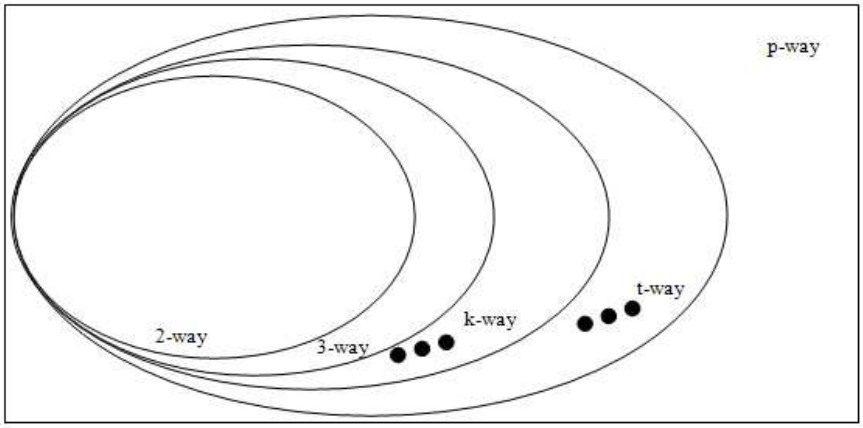

Figure 4.1: Rough Estimate of Guaranteed Coverage of Higher Order Tuples Using k-Way Tuples

stopping criteria will be discussed later in this chapter. The idea is to find a k that can guarantee coverage of as many t-tuples and other higher level tuples as possible, and not exceed the resource allocation. Any remaining tests can then be followed by random testing, which can cover additional higher order tuples that k-wise testing misses.

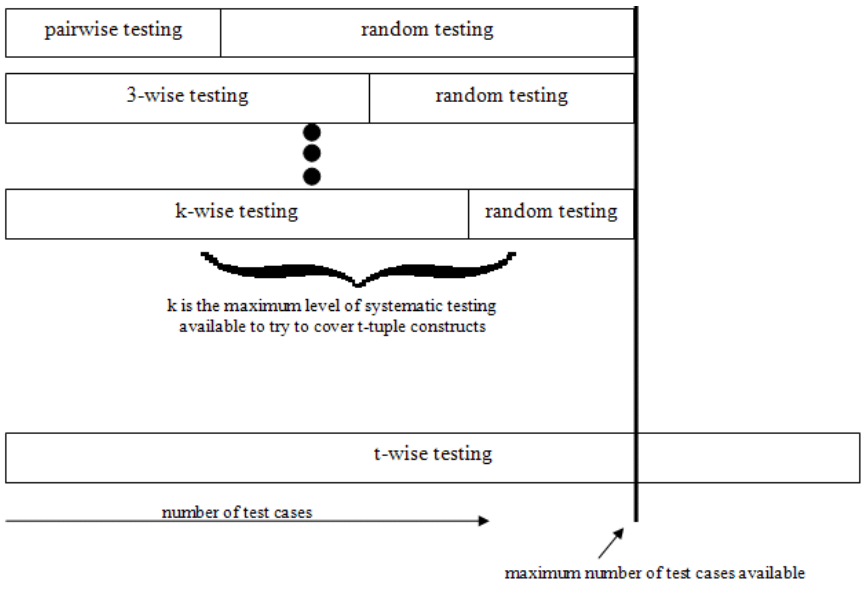

Figure 4.2: Concept of Hybrid Testing used when Full t-Tuple Testing may not be Feasible

The basic constructor for the hybrid approach is to begin with a systematic approach, namely N-wise testing, followed by random testing with replacement. One would have to decide the tuple size of the defects which would be sought (also known as t). One would also have to

assume that although all t-tuple defects are desired, one may not have the resources (the startup generation cost and the number of test cases generated) to conduct full t-tuple testing to detect those t-tuple defects. The assumptions mean that a smaller k may be required where $2 \leq k \leq t$. A method to determine the highest k is discussed later. In this section, it is assumed that $k = 2$, (i.e., pairwise testing is the systematic testing of choice). The total number of test cases in this section is based on the number of test cases of the N-wise testing approach, and the number of random test cases that have better than even chance of detecting at least one more defect. Once the largest level k-wise testing is performed, the remaining test cases are dedicated to random testing. Figure 4.2 illustrates the general concept for attempting to find t-tuple defects when t-wise testing cannot be performed.

4.2.1 Assumptions and Limitations

Some assumptions that are associated with the discussion of how to optimize the hybrid approach are as follows:

- The operational profile of the system under consideration is given (operations and frequency).
- For assessment purposes, the number of defects (m) of tuple size t in the system is known.
- $kllp$. (When $k = p$, this is exhaustive testing.)
- When a defect of a particular tuple size occurs in a particular test, the actual interacting parameters involved are not known. For example, if in the case when there are 100 parameters and there are 3-tuple defects, then there are $C(100, 3) = 161700$ possible combinations of parameters that the defect could be based on. Determining the actual interacting parameters is outside the scope of this text.
- Unless otherwise specified, random testing is with replacement.
- For the purposes of this chapter, assume that fault-failure mapping is one-to-one so that in some sense a fault is equivalent to a failure in the simulation studies reported here. Related assumptions are:

  1. All parameter values in a system are discrete. [2]
  2. All parameters can be tested in parallel.

[2] If there is a continuous range of values, then discrete values can be formed using boundary conditions [6]. For example, a range may be represented by a discrete set such that one case is less than the range, one case is greater than the range, two cases are at the extrema, and two or three cases are within the range (excluding the extremes).

3. There are no infeasible paths in the software.

4.3 Methodology

The following steps outline the optimization approach of hybrid testing as defined in this dissertation. The approach combines k-wise testing with random testing.

1. Decide on the value of k, i.e., the maximum level of N-wise testing to perform when attempting to cover t-tuples for $2 \leq k \leq t$. This is determined in part by comparing the size of the k-wise testing suite to the number of random tests required to detect at least one defect with probability of more than 1/2. The maximal level is also partially based on the optimal test-suite size (See Section 4.5.)

2. Perform k-wise testing. Start with full pairwise testing. Then determine the amount of potential overlap between pairwise and 3-wise testing and adjust the amount of testing from that suite as needed. The same adjustment occurs again with 4-wise testing, and so on up to the calculated k-wise testing level, or when the amount of overlap is 100

3. Perform random testing based on the difference between the optimal (OPT) number of k-wise test cases and the actual size of the k-wise testing suite

Note that a tester has the option of starting with systematic testing followed by random testing, or starting with random testing followed by systematic testing. In this work only the first option is considered.

4.4 How Much Random Testing?

There is a point at which k-wise testing will reach diminishing returns in terms of its ability to detect complex faults. There could be a number of reasons for this. The first possibility is that k-wise testing has a fixed number of test cases depending on p, v, and k. This means that some higher order tuples will not be covered. Also k-wise test suites based on heuristics could become fairly large, potentially causing resources to be depleted. In the extreme case, k-wise testing becomes exhaustive testing (see Equation 4.1)which is generally not feasible to have in highly complex systems since the number of combinations can become astronomically large. AllSize$_{general}$ is the exhaustive suite size in the general case, and v_i is the number of values associated with the i^{th}

parameter. Equation 4.2 shows the special case when all the parameters have the same domain sizes. Suppose that in this special case the exhaustive suite size is called AllSize. Another point of diminishing returns could occur when the probability that k-wise testing generates test cases with a particular k-tuple becomes the same as the probability of that test case being generated by random testing.

$$AllSize_{general} = \prod_{i=1}^{p} v_i \tag{4.1}$$

$$AllSize = v^p \tag{4.2}$$

Suppose one wishes to cover all m t-way defects. Let x_{min} be the minimum number of random tests needed to detect a least one defect with a probability of exceeding 1/2. This number can be calculated based on the geometric distribution which was shown in Equation 3.21. Since N-wise testing approach is based on the IPO heuristic, the overall t-way level test suite may be bigger than the optimal, which can decrease the overall efficiency of the t-wise testing k is chosen equal to t. The actual size depends heavily on p, t, and v. When conducting systematic testing, if the number of t-wise test cases generated by the chose k-wise strategy exceeds x_{min}, then one may decide to either try another type of systematic testing, such as k-wise testing alone where $2 \leq k < t$ or k-wise testing coupled with some random testing (the proposed hybrid approach in this text), or perform random testing in lieu of the particular t-wise approach. It is believed that depending on the software and needs, such as repeatable tests, one may opt to at least perform k-wise testing. Random testing improves the chance of covering t-way tuples that k-wise testing misses as well as improves the chances of detecting higher order defects.

While in some practical situations random testing with replacement may be warranted, such as when one would perhaps want to use testing without replacement but it may be too costly to check for duplicates [12, 68, 65, 66]. In the case of this work, only random testing with replacement is considered.

4.4.1 Random Testing With Replacement

Algorithm FindMaxNumRandomReplace was created to represent Equation 3.21 (geometric distribution). Substituting x for M in the algorithm and solving for x allows us to check to make sure that the size of the t-tuple test suite will not exceed the minimum size of a random test

suite. This is to ensure that efficiency will not be compromised during testing. Algorithm FindMaxNumRandomReplace gives the minimal number of test cases required to exceed a particular probability - in this case 1/2. The algorithm is used in the code below. Algorithm FindMaxNumRandomReplace has an algorithmic complexity of O(p) since the maximum of x is p-1. If the t-tuple test suite is expected to exceed the number of test cases necessary to conduct random testing with the criteria mentioned, then a maximal k is chosen to conduct k-wise testing (which is smaller than t and has a test suite expected less than or equal to the size of the calculated x where $2 \leq k < t$). The total size of the hybrid suite is then based on OPT which is v^t. If OPT $= x_1 + x_2$ where x_1 is the expected size of the k-wise suite and x_2 is the remaining number of test cases dedicated to random testing, then $x_2 = OPT - x_1$. In the case where even the pairwise testing suite is bigger than the OPT, one may opt to do one of the following: perform pairwise testing alone, perform random testing alone, or perform an alternate type of systematic testing.

Algorithm FindMaxNumRandomReplace (integer x) returns x
// This algorithm returns the minimum number of test cases (x) required
// Input: x is initially 0.
// Output: The smallest x satisfying the condition is returned.

begin
    if $x > \frac{\log(1-1/2)}{\log(1-\theta')}$ then
        return x;
    else
        return FindMaxLevelReplace($x + 1$);
end

After the maximum number of test cases has been computed for random testing, one can then perform random testing using Algorithm RandomWReplace. The algorithmic complexity is $O(p \times M)$ is obtained since the number of tests is M. Since it is in a "for" loop, this number of multiplied by $p + 1$ which is $O(p)$. Altogether, it is $O(p \times M)$.

Algorithm RandomWReplace (integer numTests, U) returns U
// This algorithm returns a set of random tests (with replacement) of size numTests.
// Input: numTests is x_{max} based on random with replacement. U is the current test suite

```
// suite. When beginning with random testing, U is null.
// Output: A random test suite, U of initial size + numTests.

begin
        for (i = 1 to numTests) do // O(M)
        begin
                Pick a random test case, rtc. // O(p)
                Add test case rtc to U. // O(1)
        end

        return U
end
```

4.5 Deciding the Maximum Level of k-wise Testing

By this point, the number of test cases has been determined by OPT (v^t). Generally, Equation 4.3 gives the maximum test suite size that any N-wise testing heuristic should give. This formula was constructed based on the worst case scenario where only one new tuple appears in any one test case, except for the very first test case (which has all unique tuples). In other words, every test case, with the exception of the very first one repeats all values except one.

$$\sum_{i=0}^{k_{max}} C\left(p-i-1, k_{max}-i\right) \times v^i \leq x_{max} \tag{4.3}$$

In the worst case scenario, one would find the largest k (called k_{max}) such that Equation 4.3 holds without compromising efficiency (i.e., it should be at least as efficient as random testing whose test suite size is represented by x_{max}). One may opt to use Equation 4.3 when unsure of the resulting test suite size from a particular heuristic used for N-wise testing. During testing data construction, one may find that the actual value of k may be lower than k_{max}, but this number can give a reference point for the maximum level of systematic testing to be performed. Algorithms CalculateKWiseSizeplace and CalculateMaxK in this section are used to represent Equation 4.3. Algorithm CalculateKwiseSizeplace has an algorithmic complexity of $O(k)$ from the 'for' loop which repeats k times and assumes the sum can be computed in O(1). Algorithm CalculateMaxK

has an algorithmic complexity of $O(k \log p)$ since the CalculateKwiseSizeplace, which is O(p), will be computed $O(\log p)$ times recursively.

In the case of the Generalized-In-Parameter-Order algorithm shown later in this chapter, the size of the test suite can be calculated using Equation 4.4. Consequently, k_{max} could also be used as a metric to determine the realistic maximum tuple sizes of defects that can be covered in the most effective and efficient manner possible.

$$v^{k_{max}} + v^{(k_{max}-1)} \log_v (p - k_{max}) \leq x_{max} \tag{4.4}$$

Algorithm CalculateKWiseSizeplace (integer k, integer p, integer v) returns sum
// This algorithm returns the absolute maximum size of a N-wise testing suite.
// Input: k is the level N-wise testing to perform, p is the number of parameters,
// v is the number of values per parameter.

begin

Set $sum = 0$. // O(1)

for $i = 0$ to k do // O(k)
begin

$sum = sum + C(p - i - 1, k - i) \times v^i$ // O(1)

end

return sum // O(1)

end

Algorithm CalculateMaxK (integer k, integer p, integer v, integer x, integer prevK) returns prevK
// This algorithm returns the k for the maximum level of k-wise testing to perform.
// Input: k is the maximum level N-wise testing to perform and is initially $\lceil p/2 \rceil$,
// p is the number of parameters, v is the number of values per parameter, x is the number of random tests,
// prevK is the stored value of the previous k.

begin

if ($k < 2$) // O(1)

```
                return 0

        sum = CalculateKWiseSize(k, p, v) //O(k)

        if(sum ≥ x) then // O(1)
        begin
                if (prevK != 0) then// O(1)
                        return prevK
                else
                        return CalculateMaxK (((0+(k−1))/2, p, v, x, prevK)) // O(log p)
        end

        else
                return CalculateMaxK (((k+1)+(p−1))/2, p, v, x, k)) // O(log p)
end
```

4.6 Performing N-wise Testing

Now that we have discussed the overall framework for combining systematic testing with random testing, we will discuss how N-wise testing will take place. Our discussion begins with giving a generalized approach based on the In-Parameter-Order algorithm given for pairwise testing [49]. The Parameter-In-Order algorithm stems from the modified N-wise testing strategy presented by Cohen et al. [19, 17]. The Generalized-In-Parameter-Order (GIPO) is as seen below.

```
Strategy Generalized-In-Parameter-Order (integer k)
// This strategy ends up with test suite, T by the end.
// Input: k is the level of testing to perform. It guarantees that all value combinations of
// k parameters at a time will appear in the test suite.
// k is typically an integer such that 2 ≤ k < p where p is the maximum number of parameters
// in the system.

begin
```

{for the first k parameters $P_1, P_2, \ldots, P_k$ respectively}
T := $\{(v_1, v_2, \ldots, v_j, \ldots, v_k) \mid v_j$ is a
value of P_j and $1 \leq j \leq k\}$

if $(k = p)$ then stop

{for the remaining parameters}
for parameter $P_i | i = k+1, k+2, \ldots, p$ do
begin
    {horizontal growth}
    for each test $(v_1, v_2, \ldots, v_k, \ldots, v_{i-1})$ in T do
        replace it with $(v_1, v_2, \ldots, v_k, \ldots, v_{i-1}, v_i)$
        where v_i is a value of P_i

    {vertical growth}
    while T does not cover all k-tuples among P_i and all (k-1) tuples of
    $P_1, P_2, \ldots, P_{i-1}$ do
        add a new test for $P_1, P_2, \ldots, P_{i-1}, P_i$ to T
end
end

Algorithm GIPO-V(π, k, T)
// {π is the set of missing tuples according to a test set T for
// $P_1, P_2, \ldots, P_{i-1}, P_i$. This algorithm constructs a minimum
// test set $T\prime$ such that T union T is a N-wise test set for
// $P_1, P_2, \ldots, P_k, \ldots, P_{i-1}, P_i$.}
// Input: π is the collection of missing k-tuples. T is the collection
// of test cases already generated.

begin
    let $T\prime$ be an empty set

for each k-tuple $\{(v_{\alpha_1}, v_{\alpha_2}, \ldots, v_{\alpha j}, \ldots, v_{\alpha k}) \,|1 \leq j \leq k; 1 \leq \alpha_j \leq i-1; v_{\alpha j} \in P_{\alpha j}\}$ in π do //
begin
if $T\prime$ contains a test t with "v_i" as the value of P_i and "-" as the value
of all the $P_{\alpha j}$ where $1 \leq j \leq k$ and $1 \leq \alpha_j \leq i-1$
then modify t by replacing all k-1 "-"s with v_{α_j}
else
add a new test to $T\prime$ that has v_i as the value of P_i, v_{α_j} as the
value $\forall P_{\alpha_j}$, and the "-" as the value of every other parameter
end
end

Algorithm GIPO-H-IV(k, *T*,P_i)
// *T* is a N-wise test set for parameters $P_1, P_2, \ldots, P_{i-1}$. This algorithm modifies
// T by adding a value of P_i to each test in *T*.
// Input: k is the level of testing. *T* is the collection of test cases already generated.
// P_i is the new parameter which will have a value added to *T*.

begin
{initialization}
π = {all k tuples of P_i with any $(k-1)$ tuples among $P_1, P_2, \ldots, P_k$,
$\ldots, P_{i-1}$}
s = min(|*D* (P_i) |, |*T*|)

for j = 1 to s do
begin
extend the j^{th} test in *T* by adding the j^{th} value of P_i
π := π - k-tuples covered by the extended test
end

if s = |*T*| then return

{necessary to select values of P_i}

for j = (s + 1) to $|T|$ do
begin
$\pi\prime$ = $\{\}$
for each value v in P_i do
begin
$\pi\prime\prime$ = {k-tuples in π covered by adding v to the j^{th} test in T}

if $|\pi\prime\prime| > |\pi\prime|$ then
begin
$\pi\prime = \pi\prime\prime$
$v\prime$ = v
end
end

extend the j^{th} test in T by adding value $v\prime$
$\pi = \pi - \pi\prime$
end
end

4.6.1 Modified N-wise Testing

To fit the proposed strategy, the algorithms described in Section 4.6 must be modified in order to optimize both test generation costs, and keep the test suite sizes as close to OPT as realistically possible based on the IPO algorithm. Consider the modified strategy of the GIPO algorithm, which is called M-GIPO (Modified-General-In-Parameter-Order). Below lists the needed modifications :

1. If there are any unassigned parametric values left in the test suite which can be any value, replace them with the latest assigned $value + 1$ modulo v in the test suite, i.e., assign it to the next value corresponding to the last selected value.

Strategy Modified-Generalized-In-Parameter-Order(integer k, integer numTests, testSuite T)
// This strategy modifies the GIPO strategy and ends up with a larger test suite, T by the end.

// Input: k is the level of testing we will perform. It guarantees that all value combinations of
// k parameters at a time will appear in the test suite.
// k is typically an integer such that $2 \leq k \leq p$ where p is the maximum number of parameters
// in the system. T is produced from a previous level of testing - either random testing or by
// $k\prime$-wise testing where $2 \leq k\prime < k$. The numTests is the calculated maximum number of test cases
// to add using N-wise testing.
begin

π = {All k-tuples not covered by T}
$T\prime$ = an empty set which will house test cases to add to T
$\pi\prime$ = an empty set which will house used k-tuples

{for the first k parameters $P_1, P_2, \ldots, P_k$ respectively and while the number of tests
has not been exceeded}
$T\prime = T\prime \cup \{(v_1, v_2, \ldots, v_j, \ldots, v_k)\}$
such that
{v_j is a value of $P_j, 1 \leq j \leq k$ and either $v_j \in \pi$ (if it exists) or
v_j = last v_j from T + 1. }

$\pi\prime$ = {All new k-tuples hit in $T\prime$}
$\pi = \pi - \pi\prime$

if (($k = p$) OR (the number of tests has been met)) then
begin
$T = T \cup T\prime$
stop
end

{for the remaining parameters}
for parameter $P_i | i = k+1, k+2, \ldots, p$ do
begin
{horizontal growth}
for each test $(v_1, v_2, \ldots, v_k, \ldots, v_{i-1})$ in $T\prime$ do

begin

replace it with $(v_1, v_2, \ldots, v_k, \ldots, v_{i-1}, v_i)$

where v_i is either a value of π or v_j = last v_j from $T\prime$ + 1.

$\pi = \pi -$ {all new k-tuples covered by $T\prime$}

end

{vertical growth}

while $T\prime$ does not cover all k-tuples in π among P_i and all (k-1) tuples of $P_1, P_2, \ldots, P_{i-1}$ and the number of test cases has not been met do

begin

add a new test for $P_1, P_2, \ldots, P_{i-1}, P_i$ to $T\prime$

$\pi = \pi -$ {all new k-tuples covered by $T\prime$}

end

end

$T = T \cup T\prime$

end

4.7 Notes on Performance of the Hybrid Approach

The purpose of this section is to discuss the performance of the hybrid approach, beyond its guaranteed bounds. In the context of hybrid testing effectiveness and efficiency are tightly coupled. Effectiveness is typically measured in terms of the number or fraction of remaining defects. But, at times it is equivalent to amount of guaranteed coverage for some particular tuple size. This can then also be used to gauge its effectiveness. Efficiency is measured in terms of the number of test cases executed.

If a testing scheme is effective in creating test cases which can potentially disclose many defects but is inefficient, it may not be useful when resource constraints are high. If a scheme is efficient in terms of effort, time and/or cost but is ineffective, it may also not be useful as a practical quality and security tool.

Coverage is often used in some form or another when testing software (e.g. [21, 50, 54]). Guaranteeing some type of structural, functional or input space coverage can aid in creating an

effective test suite for the faults that can be discovered through coverage of the constructs. In the case of N-wise testing, full coverage of all N-way tuples guarantees discovery of all N-way faults and can grant some guarantee that test cases will be generated that can potentially expose other defects. What is of interest is: "How effective is hybrid testing in detecting these "other" faults?" Obviously, by chance is one answer. What about through (incidental) coverage of higher-level N-tuples?

4.7.1 Coverage Metrics

In the context of k-wise hybrid testing, coverage of k-tuples, and of lower order tuples, is guaranteed. N-wise testing (as used in this work) provides coverage of the test-space by sampling it without replacement, i.e., it guarantees no test cases of length p are repeated in a test suite. Random testing without replacement does the same. Random testing with replacement, on the other hand, does not guarantee that repetitions of test cases will not occur. Neither scheme guarantees that there will not be a repeat of some lower level tuples throughout a test suite, which means that duplicates of t-tuples are possible where $2 \leq t < p$. As far as coverage is concerned, testing without replacement will eventually cover many, perhaps all tuples, although this may be done in an exponential amount of time. Hybrid testing combines testing with replacement and without replacement since N-wise testing is a form of testing without replacement, making it similar to random testing without replacement. The main difference between random testing without replacement and N-wise testing is that a systematic approach is typically used to generate suites instead of "guessing" at a non-repeated test case. Obviously, sampling without replacement is desirable, but it may also be expensive to eliminate duplicates. Figure 4.3 shows an example of test effectiveness versus the number of test cases for testing with replacement and hybrid testing without random testing (which is without replacement).

Coverage of Higher Order Tuples

The hybrid approach is a testing scheme without replacement. The N-wise testing component of hybrid testing uses the GIPO method, as described in Section 4.6, and locally optimizes on coverage of tuples. For example, 4-wise hybrid testing guarantees full coverage of all pairs, triplets, and 4-way tuples since 4-wise hybrid testing subsumes all lower level tuples. This type of information is important if one wishes to guarantee that all 4-way defects could be disclosed based on the test suite. It also guarantees coverage of around $\frac{1}{v}$ 5-way tuples, $\frac{1}{v^2}$ 6-way tuples and so on.

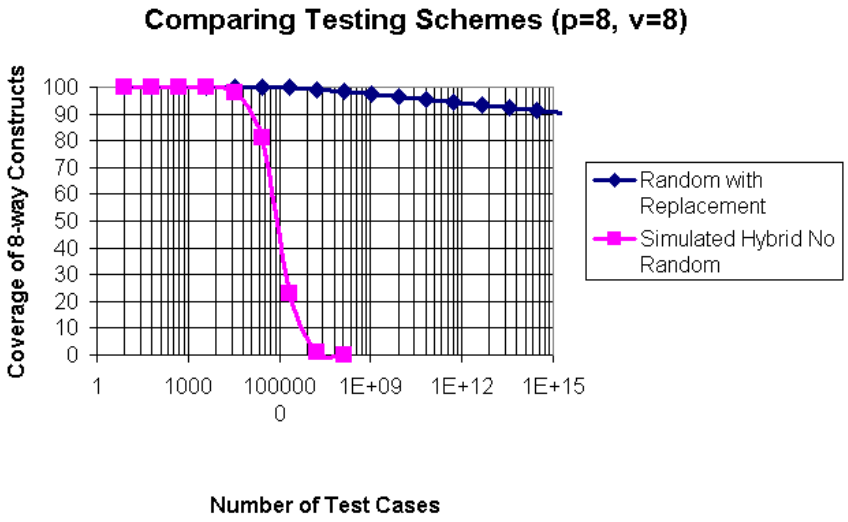

Figure 4.3: Comparison Schemes for Hybrid Testing (Minus Random Testing) and Random Testing

Such information can be used for quantitative analysis.

Figure 4.4 illustrates coverage (which in this case is synonymous with effectiveness) of four testing approaches: random, k-wise, hybrid without random, and hybrid with random when it comes to (k+1)-tuples. Also shown is ”optimal” testing strategy, i.e., one that would use only as many test cases as are needed to cover all (k+1)-way tuples in order. Note that in the case of hybrid testing, it is possible to use the N-wise (or modified N-wise) testing alone or in combination with random testing. The number of parameters is 100. The number of values is 3, and k=10, i.e., k-wise testing is performed for N-wise, and hybrid runs. This graph gives an example of the ability of the hybrid method to detect higher order tuples, specifically 11-way tuples. The hybrid without random testing uses a combination of 10-wise testing with various values for k beginning at 2 and ending at 10. In other words, pairwise testing is conducted first followed by 3-wise testing, ..., on up to 10-wise testing. For each higher k, k-way tuples subsume all smaller size tuples. For example, all pairs and triplets are subsumed during 4-wise testing.

In order to estimate coverage of higher than k-order tuples, we use the geometric distribution shown in Equation 3.21. Let H_i represent the estimated number of k-tuple constructs still left to cover at step i. Then the number of remaining k-tuples not yet covered at the i^{th} step can be estimated as seen in Equation 4.5. It is assumed that at most (k-1)-wise hybrid testing and at most k-wise testing is used to attempt to cover k-wise constructs. Let M represent the total number

of test cases run in between each step of estimating the number of uncovered tuples. M is at least 1. The initial number of k-tuples to be covered is based on the size of the space of k parameters which is v^k, i.e. $H_0 = v^k$. The C term is the number of test cases thrown out due to duplication. For random testing with replacement, none of the test cases are thrown out so C would be 0 with random testing. During the k-wise testing phase of the hybrid approach, C is a cumulative total of the total test cases already used. This number can be calculated as $i \times M$ where i is the i^{th} step of the estimations. Once the k-wise testing portion completes, C is then set to 0 for the random testing portion of the hybrid approach. Notice in the denominator, there is a $G(v)$ term. This is a function of the number of values (v), which can be based on v and the size of the space of k parameters or some other metric. In the case of Figure 4.4, the number of constructs was set to 100 so that the differences among schemes could clearly be seen.

$$H_{i+1} = H_i - H_i \times \left(1 - \left(1 - \frac{H_i}{G(v) - C}\right)^M\right) \tag{4.5}$$

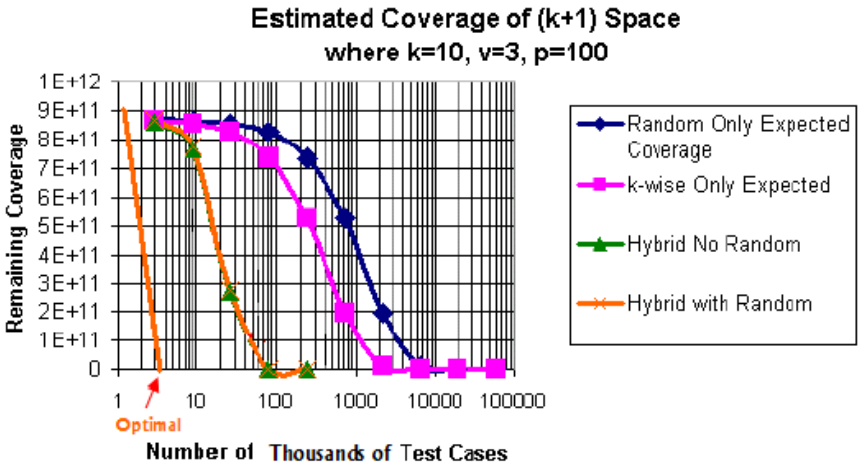

Figure 4.4: Estimated Coverage of (k+1)-way Tuples

Of course, coverage of k+1 tuple space asymptotically tends towards full coverage (but full coverage is not guaranteed and may never be reached using random methods).

Both hybrid schemes appear to be more efficient than random testing and k-wise testing alone. This graph suggests that one may choose to perform the hybrid with or without random testing to find most, if not all (k+1) tuples. The difference, however, is that the hybrid with random testing could potentially cover tuples of even higher order than the hybrid without random testing

with a fixed M may not reach.

The conclusion is therefore that there is enough evidence that the proposed hybrid algorithms add value to both plain random and plain N-wise testing which can increase problem discovery rate from several times to as much as an order of magnitude. Further work in this space is needed to compare extended N-wise and hybrid algorithms with other systematic testing methods suitable for large p and v spaces.

Chapter 5

Summary and Conclusion

The hybrid approach is a technique which merges N-wise testing and random testing techniques. N-wise testing is a particular systematic testing technique which guarantees coverage of all parametric combinations of any grouping of N parameters at least once in a testing suite. N-wise testing is useful in part because it causes the resulting test suite to remain relatively small compared to the exhaustive testing suite size with respect to the N chosen and the number of values associated with each parameter.

The N-wise testing portion of the hybrid approach is useful because it adds some guarantees about coverage that random testing does not. It also eliminates some testing redundancies at the beginning of testing, which potentially means earlier defect detection. This type of testing guarantees coverage of all value pairs between parameters, all triplets, ..., and all k-tuples among any grouping of k parameters at least once. Guaranteeing coverage may help a tester know that if a failure occurs, it could help pinpoint or eliminate possible causes assuming that test suite size affects a tester's ability to pinpoint a problem. This could decrease the cost of overall testing since it has been shown that earlier detection is equivalent to decreased software costs in general (including maintenance).

The hybrid technique first mathematically determines the maximum number of test cases overall to run and the maximum k that can be used to guarantee coverage based on the number of system parameters (p) and number of values (v). The next step is to generate the k-wise testing suite, performing pairwise testing followed by 3-wise, ..., k-wise to help ensure optimizations for the entire k-wise suite. After that completes and there are still test cases that need to be generated, random testing is performed. Ending with random testing helps to possibly cover more higher order tuples that k-wise testing missed. The technique is designed to be merged with existing processes,

such as those which employ an operational profile during random testing.

The contributions described in this dissertation are as follows:

1. There was an extension on the theory of N-wise testing. A technique was developed to generalize N-wise testing.

2. A hybrid approach was developed to help optimize effectiveness and efficiency through guaranteed coverage of pairs, triplets, ..., k-tuples to build testing suites that take advantage of lower tuple testing plus the added benefit of a certain amount of random testing to aid in coverage of missed tuples as well as higher order tuples. This approach helps decide an effective level of N-wise testing such that the size of the overall test suite minimizes hinderance on efficiency. It also takes some of the guess work away from the tester so that the test suite is not too large and has a k as practically large as possible to help cover as many complex tuples as possible. It takes advantage of the possibility of random testing possibly covering higher order tuples.

3. A semi-automatic tool was developed to help implement the hybrid approach.

Results show that the k-wise hybrid approach is at least as effective as full k-wise testing within the optimal number of test cases exercised but can cover more structures in less time because of local optimizations. The approach also seems to be at least as efficient as random testing In more complex situations, (i.e., more parameters, more values associated with parameters, and/or higher t's) the hybrid approach is more efficient than random testing. Also, the fractions of higher order t-tuples guaranteed coverage by the k-wise hybrid suite can be estimated in terms $\frac{1}{v^{t-k}}$ which could be useful for reliability analyses.

An operational profile is a historical collection of operations and frequencies of use in the field. Gittens et al. determined that information for operational profiles needed to be extended to consider other types of profiled information necessary to better meet testing needs [33]. It will be worthwhile to extend, develop, and use an operational profile which would be based in part on the tuple sizes of input and output data as well as their frequencies of occurrence. This would be to further enhance the effectiveness and efficiency of the proposed hybrid approach. This study will be conducted in a network-based environment, such as in a peer-to-peer network, because of complex interactions and complex data.

Bibliography

[1] Y. Amir, R. Caudy, A. Munjal, T. Schlossnagle, and C. Tutu. N-way fail-over infrastructure for reliable servers and routers. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 403–412, June 2003.

[2] Y. Amir, R. Caudy, A. Munjal, T. Schlossnagle, and C. Tutu. The wackamole approach to fault tolerant networks. In *Proceedings of the IEEE DARPA Information Survivability Conference and Exposition*, volume 2, pages 64–65, April 2003.

[3] K. G. Anagnostakis and M. B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, pages 524–533, 2004.

[4] M. Bailey, T. E. Moyers, and S. Ntafos. An application of random testing. In *IEEE Conference Record of Military Communications Conference*, volume 3, pages 1098–1102, November 1995.

[5] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition edition, 1990.

[6] Boris Beizer. *Black Box testing: techniques for functional testing of software and systems*. Wiley, New York, 1995.

[7] K. Z. Bell and M. A. Vouk. Assessing n-wise approach for use in system security testing. In *Supplemental Proceedings of the Fifteenth IEEE International Symposium on Software Reliability Engineering*, pages 73–74, St. Malo, France, November 2004.

[8] K. Z. Bell and M. A. Vouk. Effectiveness of stochastically generated dependencies in pairwise testing for detecting security vulnerabilities. In *Supplemental Proceedings of the Fifteenth*

IEEE International Symposium on Software Reliability Engineering, pages 33–34, St. Malo, France, November 2004.

[9] Kera Z. Bell. Random software testing: When are enhancements not necessary? In *Supplemental Proceedings of the Sixteenth IEEE International Symposium on Software Reliability Engineering*, pages 4–29 – 4–30, Chicago, Illinois, November 2005.

[10] Kera Z. Bell and Mladen A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In Mohamed A. Salem and Mahmoud T. El-Hadidi, editors, *Proceedings of the ITI Third IEEE International Conference on Information & Communications Technology: Enabling Technologies for the New Knowledge Society*, pages 221–235, Cairo, Egypt, December 2005.

[11] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the Twenty-Sixth IEEE International Conference on Software Engineering*, pages 326–335, May 2004.

[12] P. Bolaki. Empirical analysis of fault detection effectiveness of system-level n-way test suites. Master's thesis, The University of Wisconsin - Milwaukee, December 2003.

[13] QA Caf. Ip test solutions, 2005. CDRouter TestSuite.

[14] Y. Chalke, K. C. Tai, M. A. Vouk, H. Y. Chang, and Y. Lei. Pairtest version 1.1, 2002.

[15] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of the f-measure. In *Proceedings of the Fourth International Conference on Quality Software*, pages 146–153, 2004.

[16] Tsong Yueh Chen and Yuen Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, Feb 1996.

[17] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions of Software Engineering*, pages 437–444, July 1997.

[18] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (aetg) system. In *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pages 303–309, November 1994.

[19] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, pages 83–88, 1996.

[20] L. Constantin. Netwox: Network toolbox version 5.29.0, February 2005. http://www.laurentconstantin.com/.

[21] R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, Boston, 2002.

[22] Robert Culbertson, Chris Brown, and Gary Cobb. *Rapid Testing*. Software Quality Institute Series. Prentice Hall PTR, Upper Saddle River, New Jersey, 2002. www.phptr.com.

[23] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 285–294, 1999.

[24] Siddhartha R. Dalal and Allen A. McIntosh. When to stop testing for large software systems with changing code. *IEEE Software*, 20:318–323, Apr 1994.

[25] Martin Davidsson, Jiang Zheng, Nachiappan Nagappan, Laurie Williams, and Mladen Vouk. Gert: An empirical reliability estimation and testing feedback tool. In *Proceedings of the Fifteenth IEEE International Symposium on Software Reliability Engineering*, pages 269–280, November 2004.

[26] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183, 1981.

[27] E. Dustin. Orthogonally speaking. *STQE*, 3(5):46–51, September/October 2001.

[28] E-Soft. Security space security audits. http://www.securityspace.com /smysecure/index.html, 2004.

[29] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. In *Proceedings of the Third IEEE International Conference on Quality Software*, pages 63–86, January 1996.

[30] Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, and Margaret M. Burnett. Automated test generation for spreadsheets. In *Proceedings of the Twenty-Fourth IEEE International Conference on Software Engineering*, pages 141–151, 2002.

[31] H. Freeman. Software testing. *IEEE Instrumentation & Measurement Magazine*, 5:48–50, 2002.

[32] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Seventh European Software Engineering Conference*, pages 146–162, 1999.

[33] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *Proceedings of the Fifteenth IEEE International Symposium on Software Reliability Engineering*, pages 314–325, November 2004.

[34] S. D. Gouraud, A. Denise, M. C. Gaudel, and B. Marre. A new way of automating statistical testing methods. *IEEE*, 2001.

[35] Oulu University Secure Programming Group. Protos: Security testing of protocol implementations, 2004.

[36] Oulu University Secure Programming Group. Frontier-compat: Inferring causal relationships in complex systems, 2005.

[37] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, pages 4–12, 2002.

[38] M. J. Harrold. Testing: A roadmap. In *ACM Proceedings of the Conference on the Future of Software Engineering*, pages 63–72, May 2000.

[39] J. E. Heiser. An overview of software testing. In *IEEE Autotestcon Proceedings*, pages 204–211, May 1997.

[40] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, Boston, 2004.

[41] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill Book Company, New York, 1987.

[42] Tcpdump version 3.6.2, 2001.

[43] Marnie L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. Wiley Publishing, Indianapolis, Indiana, 2003.

[44] I. Ilyas, V. Markl, P. Haas, P.G. Brown, and A. Aboulnaga. Automatic relationship discovery in self-managing database systems. In *Proceedings of the IEEE International Conference on Autonomic Computing*, pages 340–341, May 2004.

[45] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Undergraduate Texts in Computer Science. Springer-Verlag, New York, 1991.

[46] Kanta Jiwnani and Marvin Zelkowitz. Maintaining software with a security perspective. In *Proceedings of the International Conference on Software Maintenance*, pages 194–203, October 2002.

[47] Edward Kit. *Software Testing in the Real World: Improving the Process*. ACM Press Books. Addison-Wesley Publishing Company, Workingham, England, 1995.

[48] D. Richard Kuhn. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[49] Yu Lei and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings of the IEEE International High-Assurance Systems Engineering Symposium*, pages 254–261, November 1998.

[50] William E. Lewis. *Software Testing and Continuous Quality Improvement*. CRC Press LLC, Boca Raton, 2000.

[51] Shin-Young Lim, Ho-Sang Hani, Myoung-Jun Kim, and Tai-Yun Kim. Design of key recovery system using multiple agent technology for electronic commerce. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, volume 2, pages 1351–1356, June 2001.

[52] Tim Menzies and Bojan Cukic. When to test less. *IEEE Software*, 17:107–112, Sep/Oct 2000.

[53] Christoph Michael and Gary McGraw. Automated software test data generation for complex programs. In *Proceedings of the Thirteenth IEEE International Conference on Automated Software Engineering*, pages 136–146, October 1998.

[54] Daniel J. Mosley and Bruce A. Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2002.

[55] John D. Musa. *Software reliability engineering: more reliable software, faster and cheaper*. Authorhouse, Bloomington, Indiana, second edition edition, 2004.

[56] John D. Musa and A. Frank Ackerman. Quantifying software validation: When to stop testing? *IEEE Software*, 6:19–27, May 1989.

[57] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill Book Company, New York, 1987.

[58] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. John Wiley & Sons, Hoboken, New Jersey, second edition edition, 2004.

[59] Goga Nicolae and Florica Moldoveanu. A distance coverage measure for bit boundary value analyse. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 973–977, October 2004.

[60] S. C. Ntafos. On comparison of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, pages 949–960, October 2001.

[61] Simeon Ntafos. On random and partition testing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48, March 1998.

[62] William E. Perry. *A Structured Approach to Systems Testing*. The QED Series. Prentice-Hall, Englewood, NJ, 1983.

[63] William E. Perry and Randall W. Rice. *Surviving the Top Ten Challenges of Software Testing: A People Oriented Approach*. ACM Press Books. Dorset House Publishing, New York, New York, 1997.

[64] Bruce Potter and Gary McGraw. Software security testing. *IEEE Security & Privacy*, pages 81–85, 2004.

[65] Anthony T. Rivers and Mladen A. Vouk. Resource-constrained non-operational testing of software. In *Proceedings of the Ninth IEEE International Symposium on Software Reliability Engineering*, pages 154–163, Nov 1998.

[66] Anthony T. Rivers and Mladen A. Vouk. Guiding resource constrained software testing. In *Proceedings of the Tenth IEEE International Symposium on Software Reliability Engineering*, pages 1–9, Nov 1999.

[67] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[68] Patrick J. Schroeder, Pankaj Bolaki, and Vijayram Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the IEEE International Symposium on Empirical Software Engineering*, pages 49–59, 2004.

[69] Patrick J. Schroeder, Eok Kim, Jerry Arshem, and Pankaj Bolaki. Combining behavior and data modeling in automated test case generation. In *Proceedings of the Third IEEE International Conference on Quality Software*, pages 247–254, November 2003.

[70] Robert C. Seacord and Allen D. Householder. A structured approach to classifying security vulnerabilities. Survivable systems, Carnegie-Mellon Software Engineering Institute, January 2005. CMU/SEI-2005-TN-003.

[71] R. Sharpe, E. Warnicke, and U. Lamping. Ethereal v0.10.5, 2004. http://www.ethereal.com.

[72] B. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 254–263, 2000.

[73] B. Smith, W. Millar, J. Dunphy, Yu-Wen Tung, P. Nayak, E. Gamble, and M. Clark. Validation and verification of the remote agent for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, volume 1, pages 449–468, March 1999.

[74] Earl W. Swokowski. *Calculus with Analytic Geometry*. PWS Kent Publishing Company, Boston, fourth edition edition, 1988.

[75] Kuo Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[76] R. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. Amsterdam: North Holland, 1978.

[77] H. H. Thompson. Why security testing is hard. *IEEE Security & Privacy Magazine*, 1(4):83–86, 2003.

[78] H. H. Thompson. Application penetration testing. *IEEE Security & Privacy Magazine*, 3(1):66–69, January/February 2005.

[79] Kishor Shridharbhai Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. Wiley, New York, 2 edition, 2002.

[80] Markos Z. Tsoukalas, Joe W. Duran, and Simeon C. Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Transactions on Software Engineering*, 19(7):687–697, July 1993.

[81] M. A. Vouk. Software reliability engineering. In *Annual Reliability and Maintainability Symposium*, pages 1–22, 2000.

[82] Mladen A. Vouk and Anthony T. Rivers. *Construction of Reliable Software in Resource-Constrained Environments*, chapter 9, pages 205–231. Wiley-Interscience, John Wiley and Sons, 2003.

[83] D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):351–371, 2001.

[84] J. A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, pages 70–79, January/February 2000.

[85] Alan L. Williams and Robert L. Probert. A practical strategy for testing pairwise coverage at network interfaces. In *Proceedings of the Seventh IEEE International Symposium on Software Reliability Engineering*, pages 269–280, 1996.

[86] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 304–311, June 2001.

[87] Jun Xu and Christopher Hoang Pham. Network vulnerability from memory abuse and experimented software defect detection. In *Proceedings of the Thirteenth IEEE International Symposium on Software Reliability Engineering*, pages 77–82, 2002.

[88] Jun Xu and M. Singhal. Cost-effective flow table designs for high-speed routers: architecture and performance evaluation. *IEEE Transactions on Computers*, 2(9):1089–1099, 2002.

[89] Shiyi Xu and Jianwen Chen. Maximum distance test. In *Proceedings of the Eleventh Asian Test Symposium*, pages 15–20, November 2002.

[90] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 45–54, July 2004.

Appendices

Appendix A

Tools to Demonstrate Hybrid Approach

PairTest is a software tool that generates a test set satisfying the pairwise testing strategy for a system [14]. The major features of PairTest include the following: - "PairTest supports the generation of pairwise test sets for systems with or without existing test sets and for systems modified due to changes of input parameters and/or values. - PairTest provides information for planning the effort of testing and the order of applying test cases. - PairTest provides a graphical user interface (GUI) to make the tool easy to use. - PairTest is written in Java and thus can run on different platforms."

The PairTest tool was developed by Dr. K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University [49]. It was later enhanced by Y. Chalke to add to it ability for automatic test case generation. It is currently going through another upgrade for use in security testing [7]. Another such tool is AETG [17]. PairTest uses a somewhat different test case generation algorithm than does AETG.

The SilverHybrid tool is an extension of the PairTest tool.

Appendix B

Extended Explanations

B.1 Utilizing Estimates of Defective Tuples

The following algorithm describes the procedure for choosing the maximum level of N-wise testing based on estimates of the number of known defective tuples.

1. Set $N = 1$. (Assume 1-wise testing is random testing here.)
2. Find the number of N-wise test cases required to minimally get a probability of more than 0.5.
3. Calculate the largest size of the test suite for (N+1)-wise testing.
4. Estimate the number of defective tuples for N-wise testing.
5. If the probability of hitting the defective tuples for N-wise testing using (N-1)-wise testing, then stop.
6. Else Set N to N+1. Goto 2.

B.2 Random Testing Followed by Optimized N-wise Testing

Once we have calculated the amount of random testing to perform and the absolute maximum level of N-wise testing (k_{max}), we can now determine how much of each test suite potentially generated from N-wise testing to perform. First, we need to determine the fraction of overlap that would occur. Since we begin with random testing, pairwise testing would follow. We would then

need to compute the amount of overlap that we would expect between the two suites. The actual amount of pairwise testing would then be dictated by choosing among the remainder fraction of pairwise test cases. Then if 3-wise testing is to be conducted, we would determine the amount of overlap from the pairwise and random testing suites, then add the remaining portion of 3-wise tests. This would repeat until we either got to the maximum level of k or we would potentially have 100% overlap. Then our test suite would be complete.

Appendix C

Parameterization of CORPORATE1 & CORPORATE2

This appendix is a collection of parameterized data for both CORPORATE1 and CORPORATE2. Each is a network-based set of data based on a collection of security failures of two particular categories identified by Security Space. The information was given in an English form and key elements were chosen to form various parameters and associated values. Please note that these lists are based on the author's expertise to parameterize information given in an online repository for one particular category [28]. The category will not be mentioned to protect the company's identity.

C.1 Login

- Number of login attempts
  - 0
  - 1
  - many
- User name types
  - long
  - invalid
  - group

  - default
  - null
  - blank
  - not applicable
- Password types
  - long
  - invalid
  - group
  - default
  - null
  - blank
  - missing
  - not applicable

C.2 Router Settings

- Enable password
  - enable
  - disable
- Cisco Express Forwarding
  - enable
  - disable
- Telnet access
  - enable
  - disable
- RTR responder

  - enable
  - disable
- IPsec over TCP
  - enable
  - disable
- SSH support
  - enable
  - disable
- IP routing
  - enable
  - disable
- Encryption
  - enable
  - disable

C.2.1 Router Authentication Settings

- Group Accounts
  - granted access
  - access denied
- User Accounts
  - granted access
  - access denied

C.3 Packet Manipulation

- Packet Form
  - Normal
  - Malformed
- Packet Inconsistency
  - layer 2 frame length mismatches encapsulated layer 3 packet
  - Not applicable
- Fragmentation
  - Fragmented
  - Not fragmented
- Frame types
  - 802.1x
- Packet Types
  - IKE
  - Response Time Responder
  - SSH
  - ICMP
  - SIP
  - ISAKMP
  - ARP
  - PPTP
- ARP types
  - request
  - reply
  - not applicable

- MAC address
  - normal
  - shorter
  - longer
- TTL
  - 0
  - 1
- Ports
  - 1720
  - 1967
  - 23
  - stp
  - 1723
  - 3100-3999
  - 5100-5999
  - 7100-7999
  - 10100-10999

C.4 Protocol Information

- Protocol
  - Swipe (53)
  - IP Mobility (55)
  - Sun ND (77)
  - Protocol Independent Multicast (103)
  - TCP
  - UDP

  - H.323

- TCP Seqnum
  - statistically generated
  - random

C.5 HTTP

- Number of clients to connect
- HTTP query types
  - Normal
  - Send a long HTTP request specifying a gateway on the local network that doesn't exist
  - Send request with filename with more than 700 bytes long
  - Send HTTP request with one character
  - Query with long readvar argument
  - http://router.address/level/NUMBER/exec/.... where NUMBER is an integer between 16 and 99
  - Send GET ?
  - Send an http request which includes /exec and a known filename (partially covered above)
  - Not applicable
- Filename
  - Exists
  - Does Not Exist
  - Not applicable
- Gateway
  - exists
  - doesn't exist

C.6 Client Information

- Number of clients
  - 1
  - 2
  - many
- Connection type
  - layer 2 with encapsulated layer 3
  - LAN-to-LAN

C.6.1 Client Options

- No encryption
  - set
  - not set

C.7 Service Information

- Services
  - telnet (TCP)
  - SSL
  - SSH
  - FTP
  - HTTP
  - SNMP (UDP)
  - Unsure
- Service requests
  - correct
  - incorrect

- Telnet options
  - None
  - Are you there (AYT)
  - Not Applicable

C.8 ACL

- Extended IP control lists - ACL
  - Keywords used in ACL
    - ∗ 'tacacs-ds'
    - ∗ 'tacacs'
    - ∗ 'established'
- ACLS options
  - exactly 448 lines
  - the last statement is not explicitly a "deny ip any any" rule

C.9 Implement Test Suite

- PROTOS test suite (for SNMP) from Ouhu University
- SSHredder test suite from Rapid 7
- Not Applicable

C.10 Miscellaneous Networking Tasks

- Announce 256 OSPF neighbors
- Specific sniffer search
  - password length
  - RSA/DSA

  - number of authorized keys
  - length of shell commands
- Miscellaneous web actions
  - View web source
- Test suites to use
  - Publicly avoidable exploits for large SSH packets (maybe covered in Rapid 7)
  - Non-commercial

Appendix D

Categories of Vulnerabilities of CORPORATE1

Table D.1: CORPORATE Category of Vulnerabilities (a)

T#	Risk	ID	Date Recorded	Formal Test Case in Form Parameter.Value
152	High	12270	6/9/2004	TCP_ACK.Invalid_Handshake
151	High	12199	4/8/2004	Packet_Types.IKE <AND> Packet_Form.Malformed
150	High	12039	2/3/2004 3/25/2004	Packet_Inconsistency.Layer_2_Frame_Length_Mismatch <AND> Switch_Type.Software
149	High	12023	1/13/2004 5/19/2004	Ports.1720 <AND> Implement_Test_Suite.PROTOS
148	High	11791	7/16/2003 8/2/2003	(Protocol.SWIPE_53 <AND> TTL.1) <OR> (Protocol.Swipe_53 <AND> TTL.0) <AND> (Protocol.IP_Mobility_55 <AND> TTL.1) <OR> (Protocol.IP_Mobility_55 <AND> TTL.0) <AND> (Protocol.Sun_ND_77 <AND> TTL.1) <OR> (Protocol.Sun_ND_77 <AND> TTL.0) <AND> (Protocol.PIM_103 <AND> TTL.1) <OR> (Protocol.PIM_103 <AND> TTL.0)
147	High	11632	5/15/2003	Packet_Form.Malformed <AND> PacketTypes.Response_Time_Responder <AND> Port.1967 <AND> RTR_Responder.Enable
146	High	11594	5/8/2003	(Packet_Form.Malformed <AND> Packet_Types.SSH <AND> IPsec_over_TCP.Enable) <OR> (Number_of_Login_Attempts.Many <AND> Packet_Form.Malformed <AND> Packet_Types.ICMP <AND> IPSec_over_TCP.Enable)
145	High	11556	4/23/2003 4/24/2003	(Service.HTTP <AND> User_Name_Types.Long) <OR> (Service.HTTP <AND> Password_Types.Long)
144	High	11547		Password_Types.Null
143	High	11383	12/16/2002	Packet_Form.Malformed <AND> Packet_Length_Specifiers.Incorrect <AND> Test_Suite.SSHredder
142	High	11382	6/27/2002	Packet_Types.SSH <AND> SSH_Options.Large_Packet <AND> SSH_Support.Enable
141	High	11381	6/27/2002	SSH_Options.Large <AND> SSH_Support.Enable
140	High	11380	2/21/2003 4/8/2004	Packet_Form.Malformed <AND> Packet_Types.SIP
139	High	11379	2/11/2003	Service.HTTP <AND> IP_Routing.Disable <AND> HTTP_Query_Types.Long_Request <AND> Gateway_Existence.Does_Not_Exist

Table D.2: CORPORATE Category of Vulnerabilities (b)

138	High	11297	9/3/2002	(Packet_Form.Malformed <AND> Packet_Types.ISAKMP) <OR> (Packet_Form.Large <AND> Packet_Types.ISAKMP)
137	Medium	11296	9/3/2002	Network_Validity.Invalid <AND> Protocol.TCP
136	Medium	11295	9/3/2002 9/4/2002	Encryption.Enable <AND> Client_Encryption.No
135	Medium	11294	9/3/2002	HTTP_Tasks.View_Source
			9/4/2002	
134	Medium	11293	9/3/2002 9/4/2002	Service.HTTP <AND> (User_Name_Types.Long <OR> Password_Types.Long)
133	Medium	11292	9/3/2002	HTTP_Tasks.View_Source
132	High	11291	9/3/2002	Group_Accounts.Access_Granted <AND> User_Accounts.Access_Denied <AND> Packet_Types.PPTP <AND> Client_Encryption.No
131	High	11290	3/28/2001	(Number_of_Login_Attempts.Many <AND> Service.SSL) <OR> (Number_of_Login_Attempts.Many <AND> Service.Telnet)
130	Low	11289	9/3/2002 9/4/2002	Service_Request.Incorrect <AND> (Service.HTTP <OR> Service.SSH <OR> Service.FTP)
129	High	11288	9/3/2002 9/4/2002	Service.HTTP <AND> HTTP_Request_Length.Long
128	High	11287	9/3/2002	Group_Accounts.Access_Granted <AND> User_Accounts.Access_Denied
127	High	11285	10/16/2002 4/20/2004	Service.HTTP <AND> HTTP_Request_Length.Long
126	High	11283	2/20/2003 2/21/2003	Packet_Types.OSPF <AND> Packet_Form.Malformed <AND> OSPF_Request_Types.Annouce_256
125	High	11056	6/27/2002 8/15/2003	Protocol.TFTP <AND> Filename_Length.More_Than_700_Byes
124	High	11014	4/9/2002	(Telnet_Access.Enable <AND> User_Name_Types.Invalid) <AND> (Telnet_Acess.Enable <AND> Password_Types.Invalid)
123	High	11012	5/9/2002 3/31/2004	HTTP_Query_Types.One_Character

Table D.3: CORPORATE Category of Vulnerabilities (c)

122	High	10987	2/12/2002 4/11/2003	
121	High	10986	7/19/2001	Service.Telnet <AND> Ports.23 <AND> Telnet_Options.Are_You_There_AYT
120	High	10985	11/28/2001	Protocol.Other
119	High	10984	11/15/2001	Packet_Types.ARP <AND> ARP_Types.Request <AND> ARP_Num_of_Requests.Many <AND> MAC_Address_Type.Different <AND> Number_Packet_Type_Requests.Many
118	High	10983	2/27/2002	Cisco_Express_Forwarding.Enable <AND> MAC_Address.Shorter <AND> IP_Layer_Length.Normal
117	High	10982	4/4/2001 9/23/2003	Protocol.NTP <AND> NTP_Query_Argument.Readvar <AND> NTP_Argument_Length.Long
116	High	10981	6/14/2001	Service.Telnet
115	High	10980	4/16/2001 4/17/2001	Ports.NotSTP <AND> Frame_Types.802.1x
114	High	10979	7/12/2001	Packet_Types.PPTP <AND> Packet_Form.Malformed <AND> Ports.1723
113	High	10978	12/6/2000 4/20/2004	Service.Telnet <AND> (<NOT> (User_Name_Types.Correct <AND> Password_Types.Correct)
112	High	10977	5/24/2001	(Protocol.TCP <AND> Ports.3100_to_3199) <AND> (Protocol.TCP <AND> Ports.5100_to_5999) <AND> (Protocol.TCP <AND> Ports.7100_to_7999) <AND> (Protocol.TCP <AND> Ports.10100_to_10999)
111	Medium	10976	5/14/2001 2/12/2003	Protocol.TCP <AND> TCP_Seqnum.Statistically_Generated <AND> Number_Packet_Type_Requests.Many
110	High	10975	8/3/2000	
109	High	10974	7/31/1995	ACL_Keywords.tacacs <OR> ACL_Keywords.tacacs
108	High	10973	6/1/1995	ACL_Keywords.established
107	Medium	10972	3/19/2001	
106	Medium	10971	11/14/2001	Packet_Types.ICMP <AND> Number_Packet_Type_Requests.Many <AND> ICMP_State.Unreachable

Table D.4: CORPORATE Category of Vulnerabilities (d)

105	High	10970	11/14/2001	ACL_Number_of_Lines.Exactly_448 <AND> ACL_Last_Statement.Is_not_deny_ip_any_any
104	High	10754		((Enable_Password.Enable <AND> Password_Types.Default) <OR> (Enable_Password.Enable <AND> Password_Types.Blank) <OR> (Enable_Password.Enable <AND> Password_Types.Missing)) <AND> ((Enable_Password.Disable <AND> Password_Types.Default) <OR> (Enable_Password.Disable <AND> Password_Types.Blank) <OR> (Enable_Password.Disable <AND> Password_Types.Missing))
103	High	10700	6/27/2001	HTTP_Query_Types.Request_with_exec <AND> HTTP_URL_Extra.NUMBER
102	High	10561	11/28/2000	Web_Administration_Interface.Enable <AND> Service.HTTP <AND> HTTP_URL_Extra.Question_Mark
101	High	10545	10/26/2000 3/30/2004	Service.HTTP <AND> HTTP_Query_Types.Send_Request_with_exec <AND> FileName_Presence.Exists

Appendix E

Bolaki's Parameterizations [12]

E.1 Parameterizing the Data Management Analysis System (DMAS)

The DMAS system was parameterized as follows: 21 input variables; a mean of 2.28 values associated with each parameter with a variance of 2.20; a medium of 2 values associated with each parameter; a minimum of 1 value associated with a parameter; and a maximum of 8 values associated with a parameter.

There were 28 indicated outputs. A mean of 4.25 interacting parameters would be required to detect a defect with a variance of 10.32. A medium of 3 interacting parameters are required to detect a defect. This is based on a minimum of 1 parameter and a maximum of 11 interacting parameters required to detect a defect.

E.2 Parameterizing the Loan Arranger System (LAS)

The LAS system was parameterized as follows: 19 input variables; a mean of 2.73 values associated with each parameter with a variance of .404; a medium of 3 values associated with each parameter; a minimum of 2 values associated with a parameter; and a maximum of 4 values associated with a parameter.

There were 14 indicated outputs. There was one output that did not have any parameters listed. It is not clear if no inputs were needed for this particular output or if the inputs were unknown. The author of this dissertation is assuming that no interacting parameters affect that particular output. A mean of 2.71 interacting parameters are required to detect a defect with a variance of 11.48. A medium of 1 parameter is required to detect a defect. This is based on a minimum of 0

interacting parameters, and a maximum of 13 interacting parameters required to detect a defect.