Software Testing: Protocol Comparison

James Yen, David Banks, P. Black, L. J. Gallagher, C. R. Hagwood, R. N. Kacker, and L. S. Rosenthal

National Institute of Standards and Technology

Gaithersburg, MD 20899 USA

March 28, 1998

Abstract: Software testing is hard, expensive, and uncertain. Many protocols have been suggested, especially in the area of conformance verification. In order to compare the efficacy of these protocols, we have implemented a designed simulation experiment that examines performance in terms of testing costs and risks. We find that no single method dominates all others, and provide guidance on how to choose the best protocol for a given application.

Software Testing: Protocol Comparison

1 Software Testing

Many researchers have proposed protocols for testing software, inspecting software, and deciding when to release software. Practitioners have also developed testing procedures, which often work well but generally lack theoretical justification. Our paper describes a methodology to resolve uncertainty about the relative merits of competing protocols, and presents results that indicate how delicate such comparisons can be.

The problem of testing software entails a tradeoff between the cost of testing and the cost of undiscovered bugs. This balance is complicated by the difficulty in estimating the number and severity of undiscovered bugs, and by the difficulty in creating realistic but mathematically tractable models for bugs. Oddly enough, some of the seminal work in this area appeared decades ago, in the context of proofreading. More recently, as software quality became a key issue and as people acquired better empirical data on software bugs, the focus has become more driven by applications.

In practice, conformance testers rely upon a straightforward, fixed-effort protocol. They generate a (sensible) test for every specification listed by the designers, then verify that the code passes all or nearly all tests. This strategy is expensive, if the specification list is long, and often fails to find interactive flaws. Nonetheless, its simplicity makes it a commonly used procedure.

A slightly different procedure is used for tests made during software development. Here one develops tests from a list of specifications. If the software fails few of these, then those bugs are corrected and the software is released. But if the software fails many of the tests, then managers meet to decide how much additional testing effort should be allocated. This protocol is a form of two-stage sampling (Cochran, 1977).

A more mathematical approach is based on Starr's model for proofreading (1974). Here one makes assumptions about the distribution of the costs of bugs, the difficulty of capturing those bugs, and then applies dynamic programming to determine when the expected cost of additional search exceeds the expected cost of the remaining bugs. Under Starr's assump-

tions, an explicit answer can be calculated; but small changes in those assumptions make closed form solutions impossible. Although the robustness of Starr's model is a key concern in practice, his approach opened up a new way of formulating these problems.

The most significant of Starr's successors is the work of Dalal and Mallows (1988). They dramatically weakened Starr's assumptions, but still obtained locally asymptotically minimax solutions. The price of this generality was an enormous increase in the mathematical difficulty of solving the dynamic programming problem; this made their solution unappealing in practice. Also, despite their unusually direct attention to reality in weakening Starr's assumptions, it remained unclear whether their solution had attractive small-sample properties.

Besides these main strategies, there are a number of other techniques for software testing. Dalal and Mallows (1998) describe tests that exercise modules or functions in pairs or triples, in the belief that most costly software errors occur as an interaction. Other workers use prior information to focus testing effort on modules that are likely to contain bugs, as in Kuo and Yang (1993, 1995). Related work, in the general context of optimal module inspection, has been done by Easterling et al. (1991).

This paper describes a simulation experiment that enables comparisons among any set of software testing protocols. Our simulation approach is generalizable, and has the potential to impact larger issues in software management, such as the maintainance of old code, the breakdown of modularity, and budgeting large projects. To illustrate the kinds of results one can achieve, we compare Starr's protocol against fixed-effort protocols that approximate current conformance testing practice. In the future, we plan to extend the simulation comparison to a more definitive set of protocols.

Section 2 describes Starr's procedure and the fixed-effort protocols that we compare. Section 3 outlines the design of the simulation experiment. Section 4 discusses our results.

2 The Protocols

This section describes the two testing protocols compared in this paper. The first approximates versions of conformance testing, as currently practiced. The second is an implemen-

tation of Starr's optimal protocol (1974), with small adaptations to accomodate estimation issues not treated in the original paper. We chose these protocols to compare because the first is simple to simulate and informs practical application, while the second represents the most mathematically sophisticated of the practicable protocols. If the latter does not offer much improvement over the former, then formal optimality properties are an unreliable guide to software validation.

2.1 Conformance Testing

In conformance testing one exercises the prescribed functionality of the software code by executing a series of tests. Usually this is done as *black-box testing*, where the tester has no knowledge of nor access to the actual code (cf. Beizer, 1995). Black-box testing is common in procurement, when a purchaser or third-party must verify that proprietary software satisfies performance specifications. At the other extreme is *white-box* conformance testing, as is often done in-house by the software developer; here the testers have complete access to the code. In between these extremes is *gray-box testing*, where some of the code is available or there is knowledge of the organization and functions of the modules (cf. Banks et al., 1998).

All three kinds of conformance testing rely heavily on problem-specific information. This dependence on context is difficult to micro-model, and thus, as a first pass, we approximate conformance testing by a fixed-effort random testing protocol. The testing effort may be heavily concentrated in specific modules (or functions) that the tester believes to be problematic, but within modules, each probe is independent of the others. From discussions with people at NIST who perform such tests (especially in the black-box scenario), there is broad agreement that the outcome from one test is roughly independent of the outcome of another. Thus the basic protocol consists of a fixed number of tests, the outcomes of which are approximately independent.

We examine six kinds of fixed-effort random testing protocols, corresponding to different levels of effort. The number of probes or tests is J, which takes the values 10, 50, 100, 500, 1000, and 5000. This range was chosen to bracket typical practice in small, non-critical software validation. For larger projects, or ones in which software failure has catastrophic consequences, one usually finds that testing is either managed independently for different

modules, or that a more complex multi-stage inspection protocol is used. Both extensions will be addressed in subsequent work.

2.2 Starr's Protocol

Starr (1974) gave an explicit solution for the problem of deciding when to stop proofreading a document, assuming that each undiscovered typographical error entails a random cost and that the proofreader is paid by the hour. For this situation, it is obvious that after a certain point, it becomes unprofitable to continue proofreading. However, finding that point entails the solution of a nontrivial dynamic programming problem.

In this paper, we develop an implementation of Starr's protocol that applies to software. The key extensions are methods to estimate the number of undiscovered bugs and the distribution of their costs, as discussed below.

Consider a set with a large (infinite) number of objects, of which N are distinguished. The problem is to identify as many as possible of the group of N objects, where there is a cost for searching and a reward for finding. The objects could be typographic errors, software bugs, or even prey. Starr assumed that the times to capture of the distinguished objects were exponentially distributed with rate μ (and mean $1/\mu$), meaning that the cumulative distribution function of the capture time X of an individual prey has the form

$$P[X < x] = 1 - e^{-\mu x}, x > 0 \tag{1}$$

and the probability density of X is

$$f(x) = \mu e^{-\mu x}, x > 0. \tag{2}$$

The value, or payoff, of the bugs upon capture are independent random variables independent of the capture times with common mean a—thus the fact that bug is hard to catch implies nothing about its payoff. Starr assumes that N, μ, and a are all known.

Although the above assumptions may seem unrealistic, they are for the most part present in the Jelinski-Moranda (J-M) model (1972), which is the most commonly used theory for analyzing software failure data. In the J-M model, the capture times for the software bugs have the same distributional form as in Starr's model. However, N and μ are not assumed

to be known, and often the payoff values are assumed to be constant. Note that the capture model implies that once a bug is "captured," it is instantaneously repaired without introducing new defects. Dalal and Mallows (1988) work with probabilistic models of software testing that are much more general and thus have the capacity to be more realistic.

Starr's analysis produced a stopping rule that is asymptotically optimal given certain assumptions. Let b be the unit cost of searching, and k_t be the number and v_t be the total value of the bugs found by time t. Starr's stopping rule maximizes expected net payoff, where the net payoff is $v_t - bt$, the total payoff of the bugs captured minus the search cost, which is b times the length of the search.

The solution maximizes the expected total payoff over all possible stopping schemes that depend only upon the present and past (thus, for example, one can't use a rule that says "Stop when the undiscovered bugs have total value less than d" because one doesn't know the value of future bugs). Starr's optimal procedure is the one that stops searching at the first time at which

$$k_t \geq N - b/(a\mu). \tag{3}$$

This optimal scheme says that one should continue searching until the number of bugs remaining in the code is no more than $b/(a\mu)$, the expected cost of capture for a single bug.

Of course, when one starts to debug software one does not know N, the actual number of bugs present, or even the rate μ or the average payoff a of the bugs. Therefore our attempt to automate Starr's procedure in a conformance testing context involves producing estimates $\hat{N}$, $\hat{\mu}$, and $\hat{a}$ after each capture, and plugging them into (1). Thus we would stop searching at the first time that the following inequality holds:

$$k_t \geq \hat{N} - b/(\hat{a}\hat{\mu}). \tag{4}$$

This approximation to Starr's rule is asymptotically valid, and its small sample properties should be very similar to those of Starr's theory.

We need now to discuss parameter estimation procedures that use only the real-time incoming data from the current software project. The obvious estimator of the average value a of the bugs is simply the arithmetic mean of the individual payoffs. But the accuracy of this estimate depends strongly upon the assumption of independence between payoff and the

length of time to catch the bug. If this is suspect, one should consider Bayesian methods, as described in section 3.

The simultaneous estimation of N and μ is problematic and has produced considerable work in the literature, most of which features maximum likelihood methods in censored or truncated situations. For example, Deemer and Votaw (1955) produce a maximum likelihood estimate of μ by considering the capture times until a time T to be observations from an exponential distribution truncated at T. However, this maximum likelihood estimator often turns out to be 0, which is impractical in our application. Similarly, in a paper on estimating population size, Blumenthal and Marcus (1975) show that the maximum likelihood estimator of N is often infinite. They attempt to alleviate that problem through a Bayesian analysis. Joe and Reid (1985) produce different modifications of the maximum likelihood methods that utilize interval estimates and harmonic means. These are the adjusted estimates in our implementation of Starr's method.

In the following discussion, we follow closely the approach of Joe and Reid (1985). Suppose that the N capture times $T_1, \ldots, T_N$ are independent and identically distributed according to an exponential distribution with rate μ. Let $t_{(1)}, \ldots, t_{(k_t)}$ be the first k_t ordered capture times. Set $w = \sum_{i=1}^{k_t} t_{(i)}/t$. Assume that $k_t \geq 1$; then Joe and Reid state that the maximum likelihood estimator of μ as a function of N is $k_t/t(w + N - k_t)$, so that the maximum likelihood estimator $\hat{N}$ is the value of N that maximizes

$$g(N|w, k_t) = \prod_{i=1}^{k_t} \frac{N - i + 1}{N + w - k_t}, \; N = k_t, k_t + 1, \ldots \tag{5}$$

Joe and Reid then divide the resulting values of the maximum likelihood estimator $\hat{N}$ into three cases:

Case 1: If $w \leq k_t / \sum_{i=1}^{k_t} (1/i)$, then $\hat{N} = k_t$.

Case 2: If $k_t \geq 2$, and $w \geq (k_t + 1)/2$ then $\hat{N} = \infty$.

Case 3: If $k_t \geq 2$ and

$$k_t / \sum_{i=1}^{k_t} (1/i) < w < (k_t + 1)/2, \tag{6}$$

then the maximum likelihood estimator is the value of k satisfying $m_{k,k_t} \leq w \leq m_{k+1,k_t}$, where

$$m_{k,r} = [1 - \{(k-r)/k\}^{1/r}]^{-1} - k + r, \tag{7}$$

for positive integers $r < k$.

Cases 1 and 2 are known as the degenerate cases. There is special trouble in Case 2, where the estimate $\hat{N} = \infty$ is useless for implementation of Starr's procedure, as it leads to the estimate $\hat{\mu} = 0$. Joe and Reid produce modified estimators of N that circumvent this difficulty. These estimates are based on their proposed interval estimates (N_1, N_2) of N, where

$$N_1(c) = \inf[N \geq k_t : g(N|w, k_t) \geq cg(\hat{N}|w, k_t)] \tag{8}$$

and

$$N_2(c) = \sup[N \geq k_t : g(N|w, k_t) \geq cg(\hat{N}|w, k_t)] \tag{9}$$

for a given c between 0 and 1. Joe and Reid advocate combining the interval endpoints N_1, N_2 in a harmonic mean to produce an estimate of N of the form

$$\tilde{N} = [\tfrac{1}{2}(N_1^{-1} + N_2^{-1})]^{-1}. \tag{10}$$

For Case 1, $N_1 = k_t$, so that $\tilde{N} = [\tfrac{1}{2}({k_t}^{-1} + N_2^{-1})]^{-1}$. and for Case 2, $N_2 = \infty$, leading to $\tilde{N} = 2N_1$.

In our implementation of Starr's procedure, we use this harmonic mean estimator $\tilde{N}$ as our estimator $\hat{N}$ for the degenerate cases but use the maximum likelihood estimator in the nondegenerate case. Then, $\hat{N}$ is used to produce an estimate $\hat{\mu}$ of μ by substituting $\hat{N}$ for N in the maximum likelihood formula $N = k_t/t(w + N - k_t)$. Simulations show that these estimates are quite good when k_t is large and is a substantial proportion of N. As would be expected, the performances of these estimators are greatly diminished when k_t is smaller in number and when it is a smaller proportion of N.

These methods for estimating the unknown parameters needed to generate Starr's stopping rule are not entirely satisfactory. Although testers do not know the exact values of N, μ, and a, they often have considerable prior information about these parameters from similar software projects that have been completed in the past. Also, experienced programmers have

a good feel for the approximate rate of bugs present and their relative costs. These considerations indicate a sequential Bayesian approach for updating prior information as data on the current software project become available. In particular, use of prior information in estimating a, the average value of the bugs, is especially important. One may have many minor bugs but a few important bugs. If, at the beginning, the discovered bugs are all minor, then a procedure that estimates a without using prior information tends to underestimate the average payoff. This error leads Starr's procedure to premature and potentially disastrous termination. In addition, it may be difficult to assess the value of each bug as it is found; in contrast, the estimation of N and μ requires only the tabulation of the bugs caught and their capture times. Starr's procedure should benefit from a Bayesian estimate of a.

Starr's research was influential, leading Dalal and Mallows (1988) to generalize the results in ways that were more faithful to the conformance testing paradigm. In particular, they allowed the costs of the bugs to be time dependent, they dropped the distributional assumptions on the length of time needed to find the bugs, and they obtained asymptotic results suggesting that the method applies when code is of inhomogeneous quality. However, it remains unclear whether their asymptotic results enable testers to achieve significant cost reductions in the finite-horizon reality of standard software testing.

3 Simulation

We need to assess competing protocols for software testing. Previous discussion has referenced the most promising strategies available in the literature. Here, we concentrate on comparing Starr's protocol with several versions of fixed-effort random testing.

Ideally, one wants an empirical comparison. By applying each testing protocol across the same set of real software with known bugs and comparing the results, a definitive evaluation becomes easy. But this approach is too expensive. Thus we replace empirical evaluation with simulation. The plan is to simulate different kinds and intensities of bugs in code, then apply each protocol and compare their success. The parameters that govern bug characteristics are chosen to match the available empirical data and expert opinion.

We emphasize that the simulation does not produce pseudocode that might actually

compile; rather, it creates lists of mathematical objects that are abstract representations of bugs and their characteristics, together with enough contextual information that the protocols can operate. Specifically, the simulation creates a list of records, each of which represents a single bug. The attributes of the record indicate various characteristics of that bug. The characteristics used in our simulation are:

Module: This indicates in which module or modules the bug occurs. (We may seek to simulate design structures among the modules as well, e.g. linear, tree, or parallel, but this aspect is secondary.)

Capture Status: A 1 if the bug has been captured and 0 otherwise.

Catchability: This is a score that measures how hard the bug is to discover.

Payoff: This indicates the reward for finding and repairing that particular bug or the cost to the producer or tester if the bug is not found.

In future work additional fields will be added, such as the probability that correction introduces a new bug and pointers to other bugs in the same cluster, so that if one bug is found, others in its clusters can have their Catchability scores revised.

A debugging process is simulated using the above quantities and each of the inspection protocols. All the capture times in this simulation take the form of the number of discrete *search passes* that have been performed on the bug list. To conduct a single search pass through a particular module, one selects one of the bugs in the module at random. If that bug has not been captured (found and fixed), then a uniform random number is generated. If that random number is less than the catchability score C for that bug, then that bug is considered to have been discovered; otherwise, that search step fails.

The complete fidelity of the simulation to a real debugging procedure may be less important than the generation of simulation data that is sufficiently accurate that one can assess the relative performances of different search protocols. Often qualitative differences emerge even from relatively simplistic simulations. For example, it is important for practitioners to develop a feel for the number of passes J that are needed for successful conformance testing on different qualities of software. Similarly, those who implement some version of Starr's

protocol need to understand its robustness to failure in the underlying assumptions. Neither of these gains depends strongly on the details of the simulation.

Our approach is inexpensive and extremely flexible—it can address other problems in software management than just inspection. If a user feels that the instantiation of the technique is unrealistic, it is simple to add new parameters and rerun the experiment. Other attributes can be included to make the modeling more realistic. For example, one could categorize the type of error (typographic, logical, or an interaction between modules), or model bug clusters and fix failures. The simulation we report is an instructive pilot, based upon a discrete analog of the J-M model.

The simulation experiments were run for the simple case of a single module. Future tests are planned for multiple modules, each with differing characteristics, and incorporating some simple structures for module connectivity. Represented in the simulation are the cases where there are 10, 20, 50, and 150 bugs. These cases are meant to represent a small range of programs from short, clean code to that which is relatively long and buggy. For these experiments, we used $b = 0.005$ as the unit cost of searching; the absolute value of this quantity is unimportant—what matters is its ratio to bug payoffs.

The Catchability score takes values in the unit interval. The lower the score, the less likely it is that a bug will be found. Here they are randomly generated from a Beta(p, q) distribution with three different pairs for (p, q):

1. $(p, q) = (0.5, 0.5)$, giving a symmetric U-shaped distribution
2. $(p, q) = (1, 1)$, giving the uniform (rectangular) distribution
3. $(p, q) = (10, 1)$, giving an asymmetric distribution with most of the weight concentrated near 0.

These values span a range of plausible scenarios. The first is a reasonable representation of code that is tested during manufacture, since many easily found bugs will be present; it also applies to conformance testing, when a vendor has failed to implement some functionality. The last distribution specifically models third-party conformance testing, where the easy bugs have largely been found and removed in-house before inspection.

As to the payoff values, these need to be positive numbers that can take very large values (representing critically important bugs). Here they are randomly generated from an exponential distribution with unit mean. Thus the typical ratio of a bug payoff to a unit of inspection time is 200; our domain experts consider this a reasonable starting point.

Of course, all of the above parameters, including the number of modules and bugs, can be adjusted to accomodate a very wide range of possible scenarios. In practice, a smart software company might run a preliminary simulation tailored to their circumstance in order to cost out the appropriate amount and kind of inspection need for a particular piece of code. The number and characteristics of modules, the structure of the modules, and the number of bugs in each module can be fixed or randomly generated (from a Poisson distribution, for instance). Similarly, the various catchability and cost indices do not have to be randomly generated, but can be fixed by the user to ensure that the protocol choice will be sensitive to, for example, the presence of elusive, expensive bugs.

4 Results

The simulation was replicated 10 times for each protocol and each combination of Catchability level and bug count. Thus there were 10 experiments in which 150 bugs were generated with random catchabilities drawn from a Beta(0.5, 0.5) distribution, along with independent random payoffs from an Exponential(1) distribution. Similarly, there were 10 experiments for each of the other bug counts and catchability distributions. The same catchabilities and payoffs were then used for each of the seven protocols (Starr's protocol and six levels of fixed-effort random search). This reuse of the random values across protocols is a variance reduction technique that allows more powerful inference on contrasts among the protocols.

For each replication, we measured four quantities on the performance of the protocol. The first is the net payoff, which is the payoff of the bugs captured minus the cost of the search. The second is the total cost of the bugs left uncaptured by the search. To combine the previous information, we calculated a penalized net score, which is the total payoff minus the cost of undiscovered bugs—this represents the best composite measure for most practical applications. Finally, we recorded that total search time; for the fixed-effort protocols, this

equals J, but there is large variation in search time for Starr's protocol.

These four numbers were averaged across the ten replicates to produce the numbers in the following tables. Table 1 shows the results from the experiment in which there are 10 bugs, with catchability scores drawn from the Beta(0.5, 0.5) distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	-13.86	3.85	7.07	7.63	6.54	4.72	-15.16
Omitted	1.98	6.12	2.70	1.90	0.98	0.31	0.18
Penal. Net	-15.84	-2.27	4.37	5.73	5.56	4.41	-15.34
Avg. Time	4380.3	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 1: Average performance criteria for 10 bugs with Beta(0.5, 0.5) catchability distribution.

To read Table 1, the first row shows the net payoff for each of the seven protocols. Note that Starr's protocol performed rather poorly, but that the fixed-effort protocols with $J = 100, 500$ did rather well. The second row shows the average cost of undiscovered bugs. Here Starr's protocol is more competitive, but protocols with $J = 1000, 5000$ do even better. The third row combines the information; the $J = 500$ protocol wins on this criterion. And the fourth row shows that, on average, Starr's protocol had difficulty in stopping early enough.

The other 11 tables are shown in the appendix and provide a more nuanced description. In particular, it is notable that Starr's apparently poor performance diminishes as the number of bugs increases and the difficulty of catching the bugs decreases. These results should be interpreted provisionally, but it sounds a clearly cautionary note in selecting the inspection protocol.

The performance of the Starr procedure as implemented as compared to random testing is mixed. The advantage of Starr's is that it provides a stopping time. Fixed-effort testing can be better than Starr's procedure in terms of net payoff if one knows the "right" value of J, the number of probes. The more bugs there are and the harder they are to find, the larger J should be. However, it is difficult to know an appropriate value of J *a priori*; if one chooses a bad value of J, then it is easy to do poorly, either by searching much too long

or not long enough. Starr's stopping rule as implemented does not choose the ideal search time, but it is usually close to the correct magnitude of J.

More generally, our interpretation is that the mathematical optimality results obtained by Starr do not transfer reliably into practical benefit (of course, this may partly reflect our choice of estimates used in implementing the protocol). Similarly, there is no value of J for fixed-effort testing that is universally superior. This is unsurprising—in problems of this complexity, no method can be expected to dominate all the competitors. Rather, some protocols are better for certain situations, while others excel in different situations. The value of the simulation study is that it enables practitioners to understand the factors that distinguish the situations, thereby supporting intelligent choice of the protocol for a specific application.

For the future, we shall experiment with more realistic situations across a broader range of protocols. These protocols will include versions of Dalal and Mallows' rule (1988), as well as two-stage protocols that undertake a preliminary set of probes in order to determine how many probes to allocate in the second stage. We also plan to modify the methods of clinical trials, used for approving drug release, to support inspection decisions about whether to release software.

5 Appendix: Simulation Tables

The following tables display the results from a 7×3×4 simulation experiment that compares seven inspection protocols across three levels of catchability distribution and four levels of bug counts. Results are expressed as averages of ten replicates, for four different criteria.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	8.65	3.66	11.40	13.60	15.34	13.26	-6.07
Omitted	10.35	15.85	7.91	5.46	1.72	1.30	0.64
Penal. Net	-1.69	-12.20	3.48	8.14	13.62	11.95	-6.71
Avg. Time	112.1	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 2: Average performance criteria for 20 bugs with Beta(0.5, 0.5) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	34.68	4.57	18.55	28.24	40.31	40.04	23.45
Omitted	13.01	45.06	30.89	20.94	6.88	4.64	1.23
Penal. Net	21.68	-40.49	-12.34	7.30	33.43	35.40	22.22
Avg. Time	399.0	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 3: Average performance criteria for 50 bugs with Beta(0.5, 0.5) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	113.60	4.38	20.11	35.69	96.19	110.50	111.83
Omitted	32.97	148.36	132.43	116.60	54.10	37.28	15.96
Penal. Net	80.63	-143.98	-112.32	-80.92	42.08	73.22	95.87
Avg. Time	1242.4	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 4: Average performance criteria for 150 bugs with Beta(0.5, 0.5) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	-41.55	5.92	9.87	9.88	8.73	6.17	-13.74
Omitted	1.63	5.29	1.14	0.88	0.03	0.09	0.00
Penal. Net	-43.18	0.63	8.73	9.00	8.70	6.07	-13.74
Avg. Time	10237.6	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 5: Average performance criteria for 10 bugs with Beta(1.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	-6.71	3.29	11.06	12.82	14.60	12.81	-6.89
Omitted	4.05	14.78	6.80	4.79	1.01	0.31	0.00
Penal. Net	-10.76	-11.49	4.26	8.03	13.59	12.50	-6.89
Avg. Time	4155.9	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 6: Average performance criteria for 20 bugs with Beta(1.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	25.96	4.47	17.22	27.51	43.81	43.08	27.65
Omitted	25.78	48.69	35.73	25.19	6.89	5.12	0.55
Penal. Net	0.18	-44.22	-18.51	2.32	36.93	37.97	27.10
Avg. Time	290.4	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 7: Average performance criteria for 50 bugs with Beta(1.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	94.67	4.79	18.25	35.19	99.03	114.05	112.75
Omitted	41.89	137.29	123.64	106.45	40.61	23.09	4.38
Penal. Net	52.78	-132.50	-105.39	-71.26	58.42	90.96	108.37
Avg. Time	1116.3	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 8: Average performance criteria for 150 bugs with Beta(1.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	-6.78	0.49	2.53	3.99	5.09	2.94	-16.42
Omitted	1.77	8.05	5.80	4.09	0.99	0.65	0.00
Penal. Net	-8.55	-7.56	-3.26	-0.10	4.10	2.29	-16.42
Avg. Time	2717.3	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 9: Average performance criteria for 10 bugs with Beta(10.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	8.40	1.80	3.59	5.37	13.53	13.46	-4.66
Omitted	10.62	20.18	18.19	16.16	6.00	3.57	1.69
Penal. Net	-2.23	-18.39	-14.59	-10.79	7.54	9.89	-6.35
Avg. Time	601.2	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 10: Average performance criteria for 20 bugs with Beta(10.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	29.65	0.64	4.43	7.57	25.46	31.86	22.95
Omitted	15.80	51.90	47.91	44.52	24.63	15.73	4.64
Penal. Net	13.85	-51.27	-43.47	-36.94	0.84	16.13	18.31
Avg. Time	1428.6	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 11: Average performance criteria for 50 bugs with Beta(10.0, 1.0) catchability distribution.

	Starr	$J = 10$	$J = 50$	$J = 100$	$J = 500$	$J = 1000$	$J = 5000$
Net Payoff	74.46	1.01	4.32	5.79	33.31	50.95	86.93
Omitted	53.87	144.89	141.38	139.66	110.15	90.00	34.03
Penal. Net	20.60	-143.88	-137.05	-133.87	-76.84	-39.06	52.90
Avg. Time	3524.3	10.0	50.0	100.0	500.0	1000.0	5000.0

Table 12: Average performance criteria for 150 bugs with Beta(10.0, 1.0) catchability distribution.

References

[1] Banks, D., Dashiell, W., Gallagher, L., Hagwood, C., Kakcer, R., and Rosenthal, L. (1998). Software testing by statistical methods. NISTIR-6129. National Institute of Standards and Technology, Gaithersburg, MD.

[2] Beizer, B. (1995). *Black-Box Testing Techniques for Functional Testing of Software and Systems*. Wiley, New York, NY.

[3] Blumenthal, S. and Marcus, R. (1975) Estimating population size with exponential failure. *Journal of the American Statistical Association*, **70**,913-922.

[4] Cochran, W. G. (1977). *Sampling Techniques*, 3rd ed. Wiley, New York, NY.

[5] Dalal, S. R., and Mallows, C. L. (1988). When should one stop testing software? *Journal of the American Statistical Association*, **83**, 872-879.

[6] Dalal, S. R., and Mallows, C. L. (1998). Covering designs for testing software. Submitted for publication.

[7] Deemer, W. and Votaw, D. (1955) Estimation of parameters of truncated or censored exponential distributions. *Annals of Mathematical Statistics*, **26**, 498-504.

[8] Easterling, R. G., Spencer, F. W., Mazumdar, M., Diegert, K. V. (1991). System-based component-test plans and operating characteristics: Binomial data. *Technometrics*, **33**, 287-298.

[9] Jelinski, Z. and Moranda, P. (1972) *Software Reliability Research, in Statistical Computer Performance Evaluation*, W. Freiberger, ed. London: Academic Press, 465-484.

[10] Joe, H. and Reid, N. (1985). Estimating the number of faults in a system. *Journal of the American Statistical Association*, **80**, 222-226.

[11] Kuo, L., and Yang, T. (1993). A sampling-based approach to software reliability. *Proceedings of the American Statistical Association Section on Statistical Computing*, 165-170.

[12] Kuo, L., and Yang, T. (1995). Bayesian computation of software reliability. *Journal of Computational and Graphical Statistics*, **4**, 65-82.

[13] Starr, N. (1974). Optimal and adaptive stopping based on capture times. *Journal of Applied Probability*, **11**, 294-301.