Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations

Ted Kremenek[1] and Dawson Engler[1]

Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{kremenek,engler}@cs.stanford.edu

Abstract. This paper explores *z-ranking*, a technique to rank error reports emitted by static program checking analysis tools. Such tools often use approximate analysis schemes, leading to false error reports. These reports can easily render the error checker useless by hiding real errors amidst the false, and by potentially causing the tool to be discarded as irrelevant. Empirically, all tools that effectively find errors have false positive rates that can easily reach 30–100%. Z-ranking employs a simple statistical model to rank those error messages most likely to be true errors over those that are least likely. This paper demonstrates that z-ranking applies to a range of program checking problems and that it performs up to an order of magnitude better than randomized ranking. Further, it has transformed previously unusable analysis tools into effective program error finders.

1 Introduction

Most compiler analysis has been built for optimization. In this context, analysis must be conservative in order to preserve program correctness after optimization. If analysis lacks the power to guarantee a decision is correct in a given context, then it is not acted on. Recently there has been a surge of interest in static program checking [1–7]. Here, the rules are somewhat different. First, sound tools must emit an error unless the tool can guarantee that code cannot violate the check. Insufficiently powerful analyses are no longer silent; instead they fail to suppress erroneous messages. Second, even unsound tools that miss errors make mistakes since practicality limits their (and their sound cousin's) analysis sophistication. These false reports can easily render both types of tools useless by hiding real errors amidst the false, and by potentially causing the tool to be discarded as irrelevant. Empirically, all tools that effectively find errors have false positive rates that can easily reach 30–100% [8, 4, 6, 7].

This paper examines how to use statistical techniques to manage the impact of these (inevitable) analysis mistakes. Program checking takes on different forms, but generally analysis results can be conceived as reports emitted by the analysis tool that take on two forms: (1) locations in the program that satisfied a checked property and (2) locations that violated the checked property. In this paper the former will be referred to as *successful* checks and the latter as *failed* checks (i.e., error reports). The underlying observation of this paper is that the most reliable error reports are based on analysis decisions that (1) flagged few errors (or failed checks) in total and (2) led to many successful checks. There are two reasons for this. First, code has relatively few errors — typical aggregate error rates are less than 5% [9]. We expect valid analysis facts to generate few error reports. Second, in our experience, analysis approximations that interact badly with code will often lead to explosions of (invalid) error reports. In contrast, the code containing a real error tends to have many successful attempts at obeying a checked property and a small number of errors.

This paper develops and evaluates *z-ranking*, a technique to rank errors from most to least probable based on the observations above. It works by (1) counting the number of successful checks versus unsuccessful checks; (2) computing a numeric value based on these frequency counts using the z-test statistic [10]; and (3) sorting error reports based on this number. Z-ranking works well in practice: on our measurements it performed better than randomized ranking 98.5% of the time. Moreover, within the first 10% of reports inspected, z-ranking found 3-7 times more real bugs on average than found by randomized ranking. It has transformed checkers we formerly gave up on into effective error finders. Z-ranking appears to be especially helpful for "safe" software checking analysis, where conservative analysis approximations often interact in unfortunate ways with source code.

In our experience, ranking of error reports is useful in several ways:

1. When a tool is first applied to code, the initial few error reports should be those most likely to be real errors so that the user can easily see if the rest of the errors are worth inspecting. In our experience, and from discussions with other practitioners, users tend to immediately discard a tool if the first two or three error reports are false positives, giving these first few slots an enormous importance. Empirically, z-ranking almost never propagates even a single false positive into these locations.
2. Even if the initial reports are good, in many cases a run of (say) 10-20 invalid errors will cause a user to stop inspecting the tool's output; therefore it is crucial to rank as many true errors at the top.
3. Often rules are only approximately true in that they only apply to certain contexts. For example, shared variables often do not need to be protected by locks in initialization code. Fortunately, when the checker hits a context where the rule does not hold there will be no (or few) successful checks and many error reports, allowing z-ranking to push the invalid errors to the bottom.
4. A tool may deliberately introduce approximations for scalability or speed or to check richer properties than is generally possible. If invalid errors follow the patterns that we hypothesize, then such steps can be taken with some assurance that when the gamble goes bad, the resulting invalid errors can be relegated below true errors.

We provided a cursory sketch of z-ranking in previous work [11], but did not explore it thoroughly and provided no experimental validation; this paper does both. More explicitly, this paper explores the following three hypotheses:

weak hypothesis: error reports coupled with many successes are probable errors ("success breeds confidence").

strong hypothesis: error reports coupled with many other error reports are improbable errors ("failure begets failure").

no-success hypothesis: error reports with no coupled successful checks are exceptionally unlikely errors. This is a useful special case of the strong hypothesis.

In our experiments, the weak hypothesis seems to always hold — its effect is that highly ranked errors tend to be true errors. The strong hypothesis often holds, but can be violated when errors cluster (see Section 6). Its effect is that low ranked error reports tend to be invalid errors.

We measure how well these hold up on data for three program property checkers. Our experimental evaluation uses a mixture of reports from both the Linux operating system and a commercial (anonymous) code base (referred to as "Company X"). Linux is a particularly good test since it is a large, widely-used source code base (we check roughly 2.0 million lines of it). As such, it serves as a known experimental base. Also, because it has been written by so many people, it is representative of many different coding styles and abilities. This helps guard against biasing our conclusions based on the idiosyncrasies of a few programmers. The commercial code base, which also is extensive in size, further facilitates studying the generality of our results.

The paper is organized as follows. Section 2 describes z-ranking. Section 3 describes our experimental setup and Sections 4–6 give the results of three checkers. Section 7 provides a quantitative analysis of z-ranking's performance and how it compares to other schemes. Finally, Section 8 discusses other issues and Section 9 concludes.

2 Z-Ranking

Abstractly, this paper reduces to solving a simple classification problem: given an error report, decide whether it is a true error or a false positive. More formally, let $\mathcal{P}$ be the population of all reports, both successful checks and failed checks, emitted by a program checker analysis tool. $\mathcal{P}$ consists of two subpopulations: $\mathcal{S}$, the subpopulation of successful checks and $\mathcal{E}$, the subpopulation of failed checks (or error reports). The set of error reports $\mathcal{E}$ can be further broken down into two subpopulations: $\mathcal{B}$, the population of true errors or bugs and $\mathcal{F}$, the population of false positives. Our classification problem can then be restated as follows: given an error report $x \in \mathcal{E}$, decide which of the two populations $\mathcal{B}$ and $\mathcal{F}$ it belongs to. The ideal classification system would make this decision perfectly. A realistic classification system will not be perfect, since this would imply the static analysis itself could be perfect, and typically each classification decision will have differing degrees of certainty. Ranking simply sorts error messages based on the confidence in the classification.

In general, classification can use many sources of information that seem relevant — the competence of the programmers involved, historical bug reports, and many others. In this paper, rather than using a priori knowledge of the particular system being inspected, we use the fact that the populations $\mathcal{B}$ and $\mathcal{F}$ have different statistical characteristics. From a statistical point of view, our problem becomes the following. First, determine measurable differences between $\mathcal{B}$ and $\mathcal{F}$. Second, use these measurements to classify error reports and compute the certainty associated with each classification. Third, sort the error reports based on this measurement.

There are many statistical techniques for determining differences between $\mathcal{B}$ and $\mathcal{F}$. In this paper we employ a simple one. Our underlying intuition is that error reports most likely to be bugs are based on analysis decisions that (1) generated few failed checks in total and (2) had many successful checks. Furthermore, error reports based on analysis decisions that (1) generated many failed checks in total and (2) had few successful checks are likely to be false positives. These intuitions rely not only on the fraction of checks that were successes, but also on the number of checks associated with an analysis decision as well. In simple terms, an explosion of failed checks is a likely indicator that something is going wrong with the analysis. Moreover, if we only observe a few failed checks associated with an analysis decision, even if a low number of successful checks occurred as well, we are less inclined to think that the error reports must be false positives.

We pursue these intuitions further for the purpose of ranking error reports first with an explorative example. We then formalize our intuitions using statistical tools and specify the complete z-ranking algorithm.

Lock Example Consider a simple intraprocedural `lock` program checker that checks that every call to `lock` must be eventually paired with a call to `unlock`. The program checker will likely conduct a flow-sensitive analysis of the procedure's control flow graph, attempting to examine all possible paths of execution. The checker emits a report indicating a *successful check* if an individual call to `lock` was matched by an individual call to `unlock`. If the checker encounters an exit point for the procedure before a matching call to `unlock` is found, however, it emits an error report (i.e., a *failed check*).

For this checker, decision analysis focuses around individual `lock` call sites. Consider three scenarios: (1) one `lock` call site has many failed checks associated with it and no successful ones, (2) a second `lock` call site has many successful checks associated with it and only a few failed ones and finally (3) a third `lock` call site has a few failed checks and a few successful ones (say of roughly equal proportion). The first scenario looks very suspicious; the large number of failed checks and no successful ones indicates the checked rule is violated to an unusual degree for this `lock` call site. The failed checks are likely to be false positives, and we should rank those error reports below those associated with the second and third `lock` call site. The second case, however, looks promising. The large number of successful checks suggests the analysis can handle the code; consequently the failed checks are likely real errors. Thus, we should rank these error reports high in the inspection ordering. The third population, however, is difficult to conclude anything about. Since it does not have an explosion of failed checks, the analysis tool appears not to be interacting poorly with the code, yet the few number of successes (and the equal proportion of successes and failures) does not imply that the rule definitely holds either. We should thus rank the error reports associated with the third `lock` call site between the reports for the other two.

2.1 Statistical Formulation

We now proceed to formalize the insights from the above `lock` checker example into a complete ranking algorithm. First observe that for the `lock` example we effectively divided the population of checks into three subpopulations, one associated with each `lock` call site. Although for the `lock` checker example we grouped messages by the source `lock` call site, different checkers will have similar notions that sets of successful and failed checks will be associated with a common analysis point. As a useful formalism, denote $\mathcal{G}$ to be the "grouping" operator that groups sets of successful and failed checks related in this manner together and partitions the set of reports $\mathcal{P}$. The precise specification of $\mathcal{G}$ will be tied to a specific checker.

Denote $\{\mathcal{P}_{\mathcal{G}_1}, \ldots, \mathcal{P}_{\mathcal{G}_m}\}$ to be the set of subpopulations of $\mathcal{P}$ created by $\mathcal{G}$. Consider any of the subpopulations $\mathcal{P}_{\mathcal{G}_i}$ of $\mathcal{P}$. Denote the number of successful checks in $\mathcal{P}_{\mathcal{G}_i}$ as $\mathcal{P}_{\mathcal{G}_i}.s$, and the number of failed checks as $\mathcal{P}_{\mathcal{G}_i}.f$. The total number of checks in $\mathcal{P}_{\mathcal{G}_i}$, which is the sum of these two statistics, is denoted $\mathcal{P}_{\mathcal{G}_i}.n$. Clearly the observed proportion of checks in $\mathcal{P}_{\mathcal{G}_i}$ that were successful checks is given by:

$$\mathcal{P}_{\mathcal{G}_i}.\hat{p} = \mathcal{P}_{\mathcal{G}_i}.s/\mathcal{P}_{\mathcal{G}_i}.n \ . \tag{1}$$

For brevity we will refer to $\mathcal{P}_{\mathcal{G}_i}.\hat{p}$ as $\hat{p_i}$. A naïve scheme would rank the populations $\{\mathcal{P}_{\mathcal{G}_1}, \ldots, \mathcal{P}_{\mathcal{G}_m}\}$ simply by their $\hat{p_i}$ values. However, this ranking ignores population size. For example, assume we have two populations of checks. In the first population, we observe one successful check and two failed checks, thus $\hat{p_i} = \frac{1}{3}$. In the second population, we observe 100 successful checks and 200 failed checks, thus $\hat{p_i}$ is also $\frac{1}{3}$. Clearly, the latter observed proportion is much less likely to be coincidental than the first. We thus wish to rank populations both by their $\hat{p_i}$ values and our degree of confidence in the estimate $\hat{p_i}$.

We do so using *hypothesis testing*, which is a standard statistical technique for comparing population statistics such as frequency counts. To do this, we conceptually treat the checks in $\mathcal{P}_{\mathcal{G}_i}$ as a sequence of "binary trials:" independent and identically distributed events that can take one of two outcomes. This abstraction essentially models the behavior of the program checker for population $\mathcal{P}_{\mathcal{G}_i}$ as a sequence of tosses from a biased coin that has a probability p_i of labeling a check as a "success" and probability $1 - p_i$ as a "failure."[1] Note that we do not know p_i; our estimate $\hat{p_i}$ will converge to it as the population size increases. A standard statistical measure of the confidence of the value of $\hat{p_i}$ as an estimate for p_i is its standard error (SE). If p is the "success" rate, σ^2 the variance, and n the number of observations then the SE for the success rate of a sequence of binary trials is given by [10]:

$$\mathrm{SE} = \sqrt{n} \cdot \sqrt{\sigma^2}/n = \sqrt{p(1-p)/n} \Longrightarrow \mathrm{SE}_{\hat{p_i}} = \sqrt{p_i(1-p_i)/\mathcal{P}_{\mathcal{G}_i}.n} \ . \quad (2)$$

Notice in Equation 2 that the SE takes into account the sample size $\mathcal{P}_{\mathcal{G}_i}.n$, and higher values for $\mathcal{P}_{\mathcal{G}_i}.n$ lead to lower values for the SE. The SE is often conventionally used to create confidence intervals that specify that the true value of p_i lies within a certain distance of $\hat{p_i}$ with a certain probability. Moreover in the domain of hypothesis testing, the SE can be used to test how likely a given $\hat{p_i}$ could have been observed assuming some value p_0 as the true parameter.[2] A standard hypothesis test for doing this is the *z-test* [10], which measures how many standard errors away an observed $\hat{p_i}$ is from p_0:

$$z = \frac{\text{observed} - \text{expected}}{\mathrm{SE}} = \frac{\hat{p_i} - p_0}{\mathrm{SE}} = \frac{\hat{p_i} - p_0}{\sqrt{p_0(1-p_0)/n}} \ . \quad (3)$$

Equation 3 provides a distance measure between a pre-specified population and an observed one. The SE is calculated assuming that p_0 is the true success rate, and the value computed by Equation 3, called the *z-score*, yields either a positive or negative measure of divergence of $\hat{p_i}$ from p_0. The z-test defines a statistically sound method of measuring the differences between two populations that incorporates both the observed $\hat{p_i}$ values and the population sizes. We can thus construct a p_0 such that populations $\mathcal{P}_{\mathcal{G}_i}$ that follow the weak hypothesis have large positive z-scores and those that follow the strong hypothesis have large negative z-scores. The value of p_0 thus serves as the separation point, or a relative baseline that seeks to differentiate error reports $x \in \mathcal{B}$ from reports $y \in \mathcal{F}$. Using these notions, we can now state the strong and weak hypotheses more formally:

weak hypothesis: For a population $\mathcal{P}_{\mathcal{G}_i}$ with high success rate $\hat{p_i}$ and low SE, the error reports $x \in (\mathcal{P}_{\mathcal{G}_i} \cap \mathcal{E})$ will tend to be true errors, and a proper choice of p_0 will cause the population to have a large positive z-score.

strong hypothesis: For a population $\mathcal{P}_{\mathcal{G}_i}$ with low success rate $\hat{p_i}$ and low SE, the error reports $x \in (\mathcal{P}_{\mathcal{G}_i} \cap \mathcal{E})$ will tend to be false positives, and a proper choice of p_0 will cause the population to have a large negative z-score.

Notationally we will denote the z-score for population $\mathcal{P}_{\mathcal{G}_i}$ as $\mathcal{P}_{\mathcal{G}_i}.z$. With the above definitions, population ranking using z-scores becomes straightforward.

[1] More formally, each check is modeled as a Bernoulli random variable with probability p_i of taking the value 1 (for a success). A sequence of checks is modeled using the Binomial distribution [12].

[2] In hypothesis testing, p_0 is known as the *null hypothesis*.

We choose p_0 so that the conditions of the strong and weak hypotheses generally hold, then compute the z-scores for each population $\mathcal{P}_{\mathcal{G}_i}$ and rank those populations in descending order of their z-scores. The last critical detail is then how to specify p_0. We provide two systematic methods:

Pre-Asymptotic Behavior: Consider the case where each subpopulation of checks $\mathcal{P}_{\mathcal{G}_i}$ is "small" (a precise definition of which is given in the next method for estimating p_0), implying a fairly high SE for all $\hat{p_i}$ values. In this case it is not always clear whether a population could be following the strong or weak hypotheses. Despite this problem, the average population success rate is an intuitive baseline that we can measure divergence from for all populations. This value is computed as:

$$\bar{p} = \left(\frac{1}{m}\right)\sum_{i=1}^{m}\hat{p_i} = \left(\frac{1}{m}\right)\sum_{i=1}^{m}\frac{\mathcal{P}_{\mathcal{G}_i}.s}{\mathcal{P}_{\mathcal{G}_i}.n} \ . \tag{4}$$

In Equation 4 the value m is the number of populations created by $\mathcal{G}$. The average success rate as a baseline is useful for the following reason. If we let $p_0 = \bar{p}$, then most populations will have low (in magnitude) z-scores. The populations, however, that have slightly lower SE values than their cousins and slightly lower or higher $\hat{p_i}$ values will have greater (in magnitude) z-scores, and they will be the most promising candidates that exhibit the strong or weak hypotheses because they are departing in the direction of those extremes. Choosing $p_0 = \bar{p}$ institutes ranking based on this divergence, and those populations that have diverged the most from the mean will be ranked the highest/lowest in the ranking.

Asymptotic Behavior: When the size of a population $\mathcal{P}_{\mathcal{G}_i}$ gets large, the SE value for $\hat{p_i}$ becomes reasonably low to believe the general trend we are seeing in that population. In this case, measuring divergences from the average population success rate does not adequately capture our expectations of how many true errors we expect to find, even if a $\hat{p_i}$ value for a population is substantially higher than the mean success rate. The main observation is that populations that are both large and have a fairly high success rate $\hat{p_i}$ but also have a substantial number of failed checks are likely to have a significant portion of those failed checks to be false positives. The basic intuition is that there cannot possibly be that many real errors.

We proceed to formalize these intuitions. For a population of checks, let s be the number of successes, f the number of failures, and b the number of real bugs. We define the error rate of a population of successes and failures as [9]:

$$\text{error rate} = b/(s+f) \ . \tag{5}$$

The error rate corresponds to the ratio of the number of bugs found to the number of times a property was checked. Empirically we know that aggregate error rates are less than 5% [9]. Consider an extreme case of the weak hypothesis

Algorithm 1 Z-Ranking Algorithm

```
1: APPLY: $\mathcal{G}$ to $\mathcal{P}$ to create subpopulations: $\{\mathcal{P}_{\mathcal{G}_1}, \ldots, \mathcal{P}_{\mathcal{G}_m}\}$
2: for all $\mathcal{P}_{\mathcal{G}_i}$ do
3:     $\mathcal{P}_{\mathcal{G}_i}.\hat{p} \leftarrow \mathcal{P}_{\mathcal{G}_i}.s / \mathcal{P}_{\mathcal{G}_i}.n$
4: if $\max_{\mathcal{P}_{\mathcal{G}_i}} \mathcal{P}_{\mathcal{G}_i}.n < 51$ then
5:     $p_0 \leftarrow \left(\frac{1}{m}\right) \sum_{i=1}^{m} \mathcal{P}_{\mathcal{G}_i}.\hat{p}$
6: else
7:     $p_0 \leftarrow 0.85$
8: for all $\mathcal{P}_{\mathcal{G}_i}$ do
9:     if $\mathcal{P}_{\mathcal{G}_i}.s = 0$ and using NO-SUCCESS HEURISTIC then
10:        Discard $\mathcal{P}_{\mathcal{G}_i}$
11:    else
12:        $\mathcal{P}_{\mathcal{G}_i}.z \leftarrow (\mathcal{P}_{\mathcal{G}_i}.\hat{p} - p_0) / \sqrt{p_0(1-p_0)/\mathcal{P}_{\mathcal{G}_i}.n}$
13: CREATE equivalence classes $\{E_{z_1}, \ldots, E_{z_k}\}$: $E_{z_i} \leftarrow \{\mathcal{P}_{\mathcal{G}_j} \cap \mathcal{E} | \mathcal{P}_{\mathcal{G}_j}.z = z_i\}$
14: SORT $\{E_{z_1}, \ldots, E_{z_k}\}$ by $z_i$ in descending order. Designate that order as
    $\{E_{z_{(1)}}, \ldots, E_{z_{(k)}}\}$.
15: for all $i = 1, \ldots, k$ do
16:     Inspect error reports in $E_{z_{(i)}}$ using an auxiliary ranking scheme
```

where all failures are real bugs and we have many successes. Equation 5, along with our knowledge of error rates, then reduces to the following inequality:

$$b/(s+b) \leq 0.5 \Longrightarrow s/(s+b) \geq 0.95 \Longrightarrow \hat{p} \geq 0.95 \ . \tag{6}$$

Equation 6 tells us for the extreme case of the weak hypothesis we generally expect $\hat{p} \geq 0.95$. Furthermore, we desire populations that exhibit this behavior to be ranked as being highly significant. We calibrate the ranking so that populations with this success rate (or greater) and small SE are ranked at least two standard errors away from p_0, providing the following constraint:

$$0.95 - p_0 \geq 2 \text{ SE} \ . \tag{7}$$

This calibration also causes populations with comparably small SE but smaller $\hat{p}_i$ to have z-scores less than 2, which is essentially the property we originally desired. We also desire that our SE to be adequately small so that our estimate of $\hat{p}_i$ is accurate enough to assume such asymptotic behavior. Thus our second constraint is:

$$\text{SE} = \sqrt{p_0(1-p_0)/n} \leq 0.1 \ . \tag{8}$$

Putting Equations 7 and 8 together and solving for p_0 and n at the boundary condition of SE $= 0.1$ we have $p_0 = 0.85$ and $n \geq 51$. Thus $n \geq 51$ is our asymptotic threshold and 0.85 is the p_0 we use for ranking.

Refinement: No-Success Heuristic The no-success heuristic discards error reports in a population that has no successful checks (i.e., $\mathcal{P}_{\mathcal{G}_i}.s = 0$). There are two intuitions for why it works. First, a population with no successes obviously has the lowest possible number of successes (zero), and thus the least likely to have any confidence of all. Second, and more subtly, unlike all other cases of errors, code that has no obeyed example of a checked property is logically consistent in that it always acts as if the checked rule does not apply. A final important observation is that if the code is always incorrect, even a single test case should show it. While programmers rarely test all paths, they often test at least one path, which would cause them to catch the error.

Sections 4-5 illustrate that in practice the no-success heuristic performs well, in one case we observe that employing the heuristic results in only inspecting approximately 25% of the error reports while discovering all true errors.

It is important to note that when the strong hypothesis fails to hold, this heuristic performs poorly. An example of this is given in Section 6 where error reports cluster heavily, causing the strong hypothesis to fail and populations with many true errors and no successes to appear.

Z-Score Ties: Equivalence Classes If we have two or more populations with the same z-score, we merge all populations $\mathcal{P}_{\mathcal{G}_i}$ with the same z-score value into an equivalence class E_{z_j}. The reason for this is because for ranking purposes populations with the same z-score are indistinguishable, and the reports in those populations should be ranked together.

The complete z-ranking algorithm is specified in Algorithm 1. To apply z-ranking, first we partition the set of reports $\mathcal{P}$ into subpopulations using the grouping operator $\mathcal{G}$ (line 1). Then we compute the $\hat{p_i}$ values for each population (lines 2-3). Next we compute p_0. If we have a population with 51 or more reports in it (our asymptotic behavior threshold), we set $p_0 = 0.85$ (line 7). Otherwise, we set p_0 to the average of the population success rates (line 5). We then iterate through the set of populations, discarding those that have no successful reports in them if we are using the no-success heuristic (line 9), and computing the individual z-scores for the others (line 12). We then merge the remaining populations into equivalence classes, where the equivalence class consists of all populations with the same z-score (line 13). We then order the equivalence classes in decreasing order of their z-scores (line 14) and then inspect the error reports one equivalence class at a time based on that order (lines 15-16). Within an equivalence class, we use an auxiliary ranking scheme to order the error reports, which can be a deterministic method (one candidate being the order the reports were emitted by the analysis tool) or some randomized or partially randomized scheme.

Note that the ranking depends completely on the choice of the grouping operator $\mathcal{G}$. This is the component that maps a particular property checker to the z-ranking algorithm. Sections 4–6 show various ways to map the results of different checkers to the z-ranking algorithm.

3 Experimental Setup

For our experiments, we measure checker results from two systems, both written in the C language. The first is the Linux kernel source tree, release 2.5.8, for which we possess a large collection of inspected results to validate our findings. The second is a large commercial (anonymous) source tree, referred to as "Company X."

The static analysis checker system we used was the *MC* system [8]. The *MC* system is a flexible framework to create a wide range of property checkers. Checkers consist of state machines specified by the checker writer. These state machines consist of patterns to match in the source code and corresponding actions, which can consist of state transitions in the checker. The intraprocedural `lock` checker from the previous section would use two states to track whether a lock was held and, if held, subsequently released. The state machine enters the initial state when the system recognizes a `lock` call in the flow graph. The system then traces subsequent possible paths in the flow graph, spawning copies of the state machine at forks in the graph. A call to `unlock` causes the state machine to transition to a termination state. If no call to `unlock` is discovered and an exit point in the control flow graph is reached, the state machine transitions to a termination state, but this time emitting an error. To be precise not all possible execution paths are explored because the checker only maintains limited contextual state information, and this allows some paths (and copies of the state machine) to be merged at program points where paths meet (and hence not all paths are fully explored) because they look equivalent from that point on. The analysis the *MC* system performs is approximate because it records only limited state information. This itself can lead to false positives (invalid errors). Furthermore, a rule a property checker inspects for may only hold in certain contexts, also leading to false positives.

We apply z-ranking to the reports of three checkers. The first two were designed around z-ranking: (1) a `lock` checker that warns when a `lock` is not paired with an `unlock` (Section 4) and (2) a `free` checker that warns about uses of freed memory (Section 5). The remaining checker was not designed to support z-ranking. It provides, however, enough information that we can do a crude application of it, getting a tentative feel for generality. In our experiments z-ranking is compared against three other ranking schemes: (1) optimal ranking, which simply consists of placing all true errors (bugs) at the beginning of the inspection order, (2) a deterministic ranking scheme used by the *MC* system, and (3) random ranking.

The deterministic ranking scheme uses several heuristics to order messages. The most important for our purposes is that it ranks intraprocedural errors over interprocedural, interprocedural errors by the depth of the call chain, and errors that span few lines or conditionals over those that span many. More detail can be found in [11].

Random ranking consists simply of inspecting a report at random, which is equivalent to sampling from a finite set without replacement. Sampling in this manner is modeled using the hypergeometric distribution: if b is the number of bugs and N the total number of reports, then the expected number of bugs found after n inspections is equal to $(b/N) \times n$ [12]. When comparing z-ranking to random ranking, we will plot the expected number of bugs found for random ranking as a function of the number of inspections.

For all checkers, we inspect the majority of all error reports emitted by the checker. We use the *MC* system to deterministically rank the reports, and we treat the top N messages as our inspection population. Furthermore, when using z-ranking, for inspecting reports *within* populations we employed the *MC* deterministic ranking scheme.

A key point to note is that the false positive rates for all the checkers are noticeably more pessimistic than what would actually be observed in practice. First, the false positive rates assume almost all messages are inspected. Somewhat surprisingly such diligence is rare: coders often inspect errors until they hit too many false positives and then stop. As a result, good ranking is crucial. Second, in practice putting related error into aggregated equivalence classes makes inspection more efficient since when we hit a false positive, we can suppress the entire class. Finally, many false positives are "historical" in that they were inspected at some point in the past and marked as false positives — the *MC* system automatically suppresses these in future runs, although in this paper they are counted.

4 Results: Ranking Lock Errors

This section measures the effectiveness of z-ranking on sorting lock errors. The checker takes a list of function pairs $(l_0, u_0), \ldots, (l_n, u_n)$ and, after a call to l_i, traverses all subsequent control flow paths, checking that they contain a call to u_i. As discussed in prior sections, for the purposes of z-ranking, each path where the checker encounters a u_i is considered a success; each path that ends without a u_i generates an error message and is considered a failure. The grouping operator $\mathcal{G}$ groups messages that have a common l_i call site.

The checker suffers from two general approximations. First, its limited intraprocedural world view causes it to falsely flag errors where a required call to u_i is contained in a called function or in the calling function. The second more difficult issue to remedy is the conflated roles of semaphores, which are sometimes used as counters, which need not be paired (and hence should not be checked), and sometimes as locks, which must be paired. Both limits can cause many false positives; both are handled well by statistical ranking.

We checked four pairs of locking functions; while all conceptually must obey the same rule, they form three reasonably distinct populations in Linux.

Population 1: is made of errors from the pair `spin_lock`-`spin_unlock`, which are the most widely-used lock functions (roughly 5600 checks in our results). This population has been deformed by many previous bug fixes. In the past, we have reported roughly two hundred errors involving these two functions to Linux maintainers. Most of these errors were fixed. Consequently, the ratio of true errors reported in 2.5.8 to the number of checks is low. Similarly, the

```
// linux/2.5.8/mm/shmem.c:shmem_getpage_locked
repeat:
  spin_lock (&info->lock);  // line 506
  ...
  if (...) {
    // NOTE:517: [SUCCESS=spin_lock:506]
    spin_unlock (&info->lock);
    return page;
  }
  ...
  if (entry->val) {
    ...
    if (...) {
       UnlockPage(page);
       // "ERROR:shmem_getpage_locked:506:554:": Didn't reverse 'spin_lock'
       // [FAIL=spin_lock:506]
       return ERR_PTR(error);
    }
    ...
  }
  ...
  wait_retry:
   // NOTE:597: [SUCCESS=spin_lock:506]
   spin_unlock (&info->lock);
...
// 2.5.8/drivers/ieee1394/sbp2.c:sbp2_agent_reset
  if (!(flags & SBP2_SEND_NO_WAIT)) {
   // ERROR: Did not reverse 'down' [FAIL=down:1847]
   down(&pkt->state_change); // signal a state change
```

Fig. 1. Lock check example: in the first function, `shmem_getpage_locked`, the lock `info→lock` is acquired at line 506, has five successful releases (two shown, denoted `SUCCESS=spin_lock:506`) and one error (denoted `FAIL=spin_lock:506`). The second function `sbp2_agent_reset` uses the semaphore `pkt→state_change` to atomically signal a state change rather than as a lock. There were no successes in this function, and the generated error is a false positive.

false positive rate is quite high, since while many errors have been fixed the previous false positives have not been "removed."

The two remaining populations below represent errors for the whole of the kernel's lifetime:

Population 2: is made of two straightforward pairing functions that we did not check previously. They have a higher error rate and a lower false positive rate than spin locks. They come from two pairs. First, `lock_kernel-unlock_kernel` which control the "the big kernel lock" (or BKL), a coarse-grained lock originally used (as in many initially single-threaded Unix OSes) to make Linux into a large monitor; its use has been gradually phased out in favor of more fine-grained locking. Second, the pair `cli-sti` and `cli-restore_flags`, the most widely used way to disable and enable interrupts.

Population 3: `down-up` semaphore functions, which have a high false positive rate since they have two conflated uses: as atomic counters, which need not be paired, and as locks, which must be paired. Ranking easily distinguishes

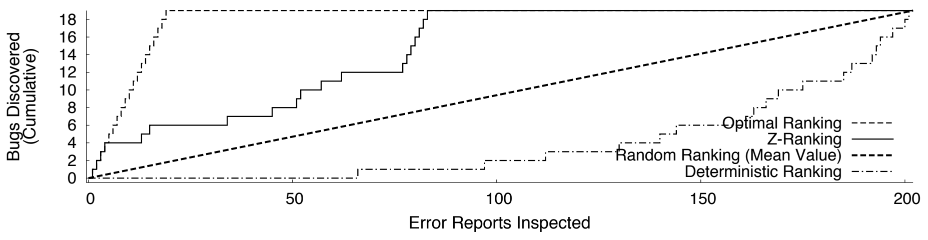

Fig. 2. Results of inspecting Linux spin-lock error reports. Z-ranking uses $p_0 = 0.21$, with 175 populations created by $\mathcal{G}$, and 14 equivalence classes. 202 reports inspected, with 19 real bugs. Within the first 21 ($\sim$ 10%) inspections, z-ranking found 3 times more bugs than the expected number found using random ranking. When using the no-success heuristic, we inspect only the first 83 reports, discovering all real bugs.

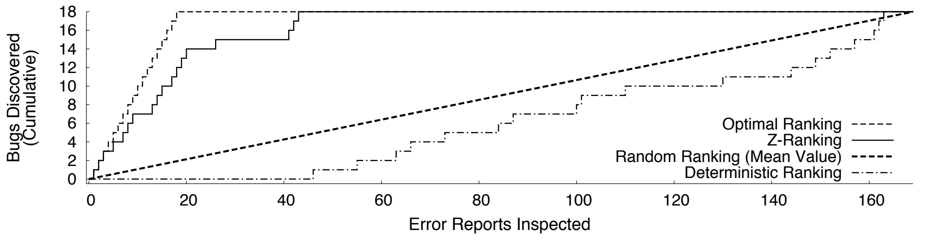

Fig. 3. Results of inspecting Linux `down-up` (semaphore) pairs. Z-ranking uses $p_0 = 0.15$, with 142 populations created by $\mathcal{G}$, and 11 equivalence classes. 169 reports inspected, with 18 real bugs. Within the first 17 ($\sim$ 10%) inspections, z-ranking found 6.6 times more bugs than the expected number found using random ranking. When using the no-success heuristic, we inspect only the first 43 reports, discovering all real bugs.

these two different uses, whereas adding additional traditional analysis will not. (In fact, we had previously given up on checking this rule since the false positive rate was unmanageable.)

Example Fig. 1 illustrates successes and failures the described checker finds in a section of the Linux 2.5.8 kernel source. An acquisition to `spin_lock` is made and depicted are two successful releases and one failed one. These checks correspond to the same `spin_lock` call site, and the grouping operator $\mathcal{G}$ would group these checks into the same population. In addition, an unmatched call to `down` is shown at the bottom of Fig. 1. This unsuccessful check would be grouped in a different population, since the check corresponds to a different source call site.

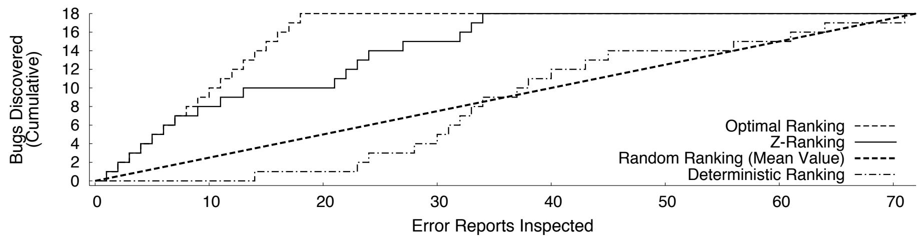

Fig. 4. Results of inspecting Linux BKL calls and interrupt enable/disable calls. Z-ranking uses $p_0 = 0.21$, with 62 populations created by $\mathcal{G}$, and 8 equivalence classes. 72 reports inspected, with 18 real bugs. Within the first 8 ($\sim$ 10%) inspections, z-ranking found 4 times more bugs than the expected number found using random ranking. When using the no-success heuristic, we inspect only the first 34 reports, discovering all real bugs.

Results We compared the use of z-ranking for inspecting error reports to both the deterministic ranking scheme in the *MC* system and randomized ranking. Depicted in Fig. 2-4 are the comparative z-ranking inspection results. To perform z-ranking, different values for p_0 were automatically estimated according to the procedure in Algorithm 1, and these values are shown in the corresponding figures. In all cases z-ranking performs well.

In the case of the `spin_lock` errors (Fig. 2), the number of inspections needed to recover all bugs in the report set is very large, but inspection using z-ranking yields numerous bugs in the first few inspections. Moreover, with deterministic ranking over 50 inspections are needed to discover the first bug. Even with random ranking a user would need to inspect on average 10 error reports before discovering a single bug. Furthermore, the no-success heuristic performs very well. After inspecting 83 of the 202 reports (41%) all bugs are discovered.

In the case of the semaphore `down` data (Fig. 3), z-ranking performs even better. Here the weak hypothesis fervently comes into play as populations where the alternate use of `down` as atomic counters instead of lock acquisitions will yield very few "successful" pairings with `up` and these error reports (which are invalid errors) will be pushed towards the end of the inspection ordering. In addition, the no-success heuristic again performs well. After inspecting 43 of 169 reports (25.4%) all the bugs are discovered.

For the remaining population of big kernel lock acquisitions and interrupt enable/disable routines (Fig. 4), z-ranking performs very close to optimal ranking for the first dozen inspections, and performs far better than random ranking and deterministic ranking. Inspection of the reports using z-ranking yields all 18 bugs within the first 38 inspections, while deterministic ranking requires inspecting 77 out of 78 reports to discover all the true errors, which is over twice as many inspections. Moreover, the no-success heuristic remains effective, requiring inspecting roughly half of the error reports to discover all the bugs.

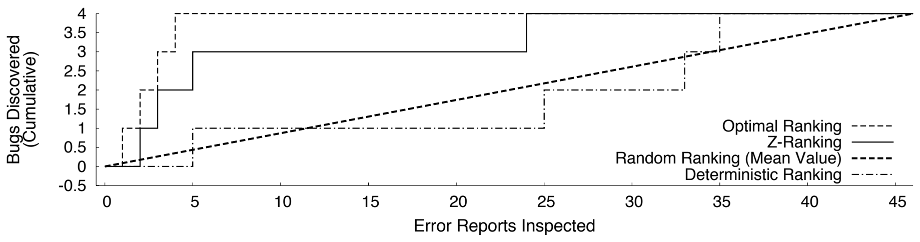

Fig. 5. Results of inspecting lock errors for Company X. Z-ranking uses $p_0 = 0.32$, with 38 populations created by $\mathcal{G}$, and 9 equivalence classes. 46 reports inspected, with 4 real bugs. Within the first 5 ($\sim 10\%$) inspections, z-ranking found 6.9 times more bugs than the expected number found using random ranking. When using the no-success heuristic, we inspect only the first 24 reports, discovering all real bugs.

In all cases, the crucial initial few error reports were always true errors. In contrast, random ranking and deterministic ranking both had a very high number of false positives in these slots (often all were false). As noted in the introduction, if these first few reports are false users often immediately give up on the tool.

Company X The results for Company X came after previous rounds of bug fixes, which deformed the population similarly to the `spin_lock` population of Linux. As a result there were only four bugs left when the checker ran, making it hard to get a good aggregate picture. The inspection results are shown in Fig. 5. Despite there being only a few remaining bugs, the results seem relatively similar to the Linux results: z-ranking places most bugs at the beginning of the inspection ordering and most false positives are at the end.

5 Results: Ranking Free Errors

The previous section used z-ranking to compensate for intraprocedural analysis approximations. This section uses it to control the impact of inter-procedural analysis approximations for a checker that warns about potential uses of free memory.

The checker is organized as two passes. The first pass uses a flow-insensitive, interprocedural analysis to compute a list of all functions that transitively free their arguments by calling a free function directly (such as `kfree`, `vfree`, etc) or by passing an argument to a function that does. The second, flow-sensitive, intraprocedural pass, uses this summary list to find errors. At every function call, it checks if the function is a free function and, if so, marks the pointer passed as the freeing argument as freed. It then emits an error report if any subsequent path uses a freed pointer.

In practice, the checker suffers from two main sources of false positives. First, false paths in the source code can cause it to think that a freed pointer can reach a use when it cannot. Second, and more serious, a small number of functions will free an argument based on the value of another argument. However, the flow-insensitive relaxation is blind to such parameter data dependencies. Thus, it will classify such functions as always freeing their argument. As a result, rather than having an error rate of one error per few hundred call sites, these functions will have rates closer to fifty errors per hundred call sites, giving a flood of false positives. Fortunately, using z-ranking to sort based on these error rates will push real errors to the top of the list and the false positives caused by such functions the analysis could not handle will go to the bottom.

We apply z-ranking to the free errors as follows:

1. We count the number of checks rather than the number of successes. For example, if `kfree` is a free function, we count a check every time we see a call to `kfree`.
2. Each error message (where freed memory was used) is a failure.
3. After the source code has been processed, the grouping operator $\mathcal{G}$ groups all checks and failures that correspond to the same free function (the function itself, not a particular call site). We then rank the populations using Algorithm 1.

The end effect is that routines with a high ratio of checks to failures will be ranked at the top and routines with low ratios at the bottom. In the Linux kernel, the routine `CardServices` is a great example of this. It has a `switch` statement with over 50 case arms selected by the first parameter. One of these case arms frees the second parameter. Our checker is too weak to detect this data dependency and, since `CardServices` can free its argument on a single path, the checker assumes it always frees its argument on all paths. Fortunately, the enormous number of (false) error reports push these reports to the lowest of all routines in the ranking, effectively eliminating them.

Interprocedural Results Our experimental results only consider routines that required interprocedural analysis. Since there were many reports we only inspected errors that involved functions that called free functions with a chain depth of less than four. We expect this result to underestimate the effectiveness of z-ranking since we expect that deeper call chains have even more false positives (since there were more opportunities for mistakes) and hence would benefit more from our technique.

The error report inspection plot for the interprocedural checker is depicted in Fig. 6. The highest ranked population was for function `netif_rx` which had 180 checks and one failure. The last (worst) ranked population was for the routine `CardServices` — it had even more false positives than shown, we stopped marking them after a while. Z-ranking does better than random ranking, though not as substantially as with the lock checker. In part this is due to the fact that there are more equivalence classes with a high number of false positives, dragging all ranking methods to the same average.

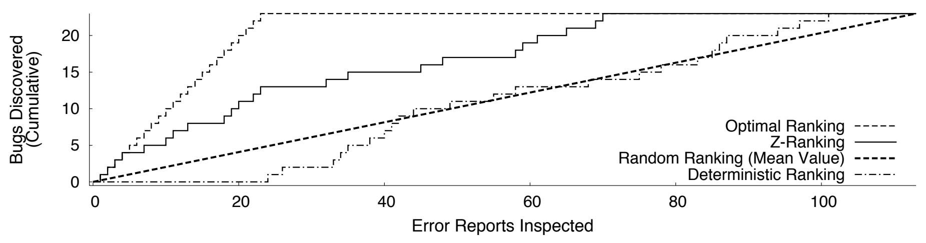

Fig. 6. Results of inspecting interprocedural free calls in Linux. Z-ranking uses $p_0 = 0.85$ (Asymptotic Behavior), with 55 populations created by $\mathcal{G}$, and 21 equivalence classes. 113 reports inspected, with 23 real bugs. Within the first 12 ($\sim$ 10%) inspections, z-ranking found 3.3 times more bugs than the expected number found using random ranking. For this set of reports all populations have at least one failure (the no-success heuristic does not apply).

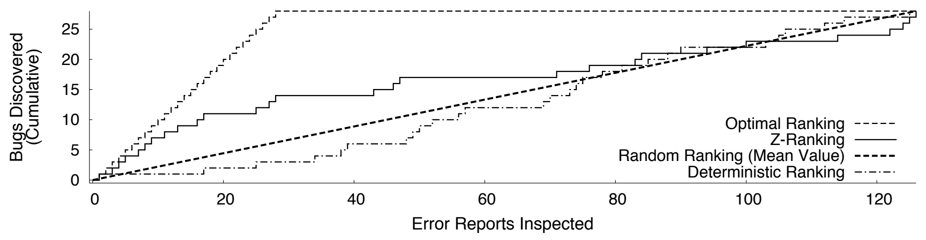

Fig. 7. Unaltered format string checker results for Company X. Z-ranking uses $p_0 = 0.29$, with 108 populations created by $\mathcal{G}$, and 18 equivalence classes. 126 reports inspected, with 28 real bugs. Within the first 13 ($\sim$ 10%) inspections, z-ranking found 3.1 times more bugs than the expected number found using random ranking. When using the no-success heuristic, we inspect only the first 60 reports, discovering 17 real bugs.

6 Results: String Format Errors

This section looks at the applicability of z-ranking even for checkers where it would appear not to apply. Although not formally conclusive, these empirical results provide insight on potential "lower-bounds" of how generally applicable z-ranking is for ranking error reports.

We use z-ranking to rank error reports for a checker that looks at security holes caused by format string errors (e.g., `printf`). The results presented were from the application of the security checker to Company X's source code base. The z-ranking results of this checker are pessimistic for three reasons:

1. The checker has not been modified to support z-ranking. For historical reasons it emits a single note for each call site that begins a check rather than

emitting a note for each success. Since a single call site can spawn many errors and only crudely correlate with successful checks, the produced check counts have significant noise.

2. The only report feature available for grouping is the name of the function the report occurred in. Thus, we group all the checks for a function into the same equivalence class. As with the previous approximation, aggregating different checks loses valuable information and can have a marked impact on the checkers from the previous section.
3. The errors are susceptible to "clustering" in that the checkers are for rules that programmers often simply do not understand and thus violate exuberantly. The end results is that in some cases, the presence of one error *increases* the probability of another. This violates our strong hypothesis. However, the weak hypothesis holds in that many checks and one failed check in a population still tends to indicate a likely error.

The inspection results for the security checker are shown in Fig. 7. Even with the above three handicaps the results are extremely encouraging. Within inspecting the first 17 error reports, z-ranking finds 11 of the 28 bugs, while inspection using deterministic ranking only finds 3. Only after around 70 checks does the performance of deterministic ranking overtake z-ranking, although at this point random ranking performs just as well.

It is surprising that despite the presence of errors clustering z-ranking does reasonably well. The weak hypothesis pushes many of the true error reports to the top. We see that after inspecting 10% of the error reports z-ranking still finds 3.1 times more bugs than the expected number found by random ranking. The clustering of errors, however, causes the strong hypothesis not to hold. This appears to have a significant effect on z-ranking's performance, as after inspecting 50% of the reports its performance matches that of random ranking. Fortunately, this performance degredation occurs at the tail of the inspection process and not the beginning. Inspection using the no-success heuristic appears particularly dismal; many real errors are missed on account of the errors clustering. In this case, z-ranking is best applied without the no-success heuristic. Moreover, were the weak hypothesis to also fail z-ranking would likely perform extremely poorly.

7 Quantitative Evaluation

In the previous sections we examined the application of z-ranking to sorting error reports in several checkers. Those results demonstrated that in most cases z-ranking dramatically outperforms randomized rankings. Z-ranking, however, rarely produces an optimal ranking. This section further quantifies the efficacy of z-ranking in the domains we examined.

An analysis tool generates a set of N error reports, where N can be hundreds of error reports. There are $N!$ possible orderings of those reports. For a given p_0, z-ranking chooses a ranking R_Z out of this large space of rankings. One way to

Table 1. Z-Ranking score $S(R_Z)$ compared to the scores of 1.0×10^5 randomly generated rankings. Column 2 lists the number of random rankings whose score $S(R_R)$ was less than or equal to $S(R_Z)$ (lower score is better). Column 3 lists the same quantity as a percentage

Checker	**Number of R_R: $S(R_R) \leq S(R_Z)$**	**Percentage (%)**
Linux `spin_lock`	0	0.0
Linux `down-up`	0	0.0
Linux BKL, interrupt enable/disable	0	0.0
Company X - `lock`	825	0.825
Linux Interprocedural Free	0	0.0
Company X - Format String	1518	1.518

quantify how good the choice of R_Z was is by asking how many other rankings R_R could have provided as good or better as an inspection ordering as z-ranking. To pursue this question, we need a quantitative measure to compare rankings. After i inspections, an optimal ranking maximizes the *cumulative number* of true errors discovered. Other ranking schemes should aspire to this goal. Let N be the total number of error reports and b the number of bugs. Let $R(i)$ denote the cumulative number of bugs found by a ranking scheme R on the ith inspection. If R_O is an optimal ranking, note that $R_O(i) = min(b, i)$ (the minimum of b and i). An intuitive scoring for R is the sum of the differences between $R_O(i)$ and $R(i)$ over all inspection steps :

$$S(R) = \sum_{i=1}^{N} [R_O(i) - R(i)] = \sum_{i=1}^{N} [min(i, b) - R(i)] \quad . \tag{9}$$

Note that Equation 9 is simply the *area* between the plots of the cumulative number of bugs found versus the number of inspections for an optimal ranking and a ranking R. Observe that $S(R_O) = 0$, so a lower score is a better score. Using $S(R)$, we can ask the question that out of all the possible $N!$ rankings, what proportion of them perform as good or better than z-ranking? For rankings consisting of greater than 10 error reports it is computationally prohibitive to enumerate all possible rankings. Instead we settle for an approximation. For each of the checkers we applied z-ranking to in this paper, we generated 1.0×10^5 random rankings. The number of random rankings that scored as good or better than R_Z (i.e., $S(R_R) \leq S(R_Z)$) is shown in Table 1. Not surprisingly, the checker with the highest number of random rankings that had a score as good or better than z-ranking was the format string checker. We recall, however, that this checker was not even designed with z-ranking in mind, and the percentage of randomly generated rankings that were better than z-ranking was only 1.518%. Moreover, Table 1 shows that in practice random ranking will rarely perform as well as z-ranking (at least according to Equation 9) for the checkers we analyzed.

8 Discussion: Possible Extensions to Z-Ranking

Z-ranking employs a simple statistical model to rank error reports. Extensions to the simple model may facilitate more sophisticated ranking schemes.

One immediate extension is including prior information into the ranking process about the source code being checked or the checker itself. Such prior knowledge could be specified by hand or possibly be determined using statistical or machine learning techniques. In both cases, one immediate approach would be to encode the prior using a Beta distribution [12], which is conjugate to the Binomial and Bernoulli distributions [13]. In this case, the prior would be represented by "imaginary" success/failure counts. These would then be combined directly with the observed success/failure counts and z-ranking could then be applied as usual on the combined counts. Using the Beta distribution also allows one to specify the "strength" of the prior by varying the number of imaginary counts; this helps facilitate fine tuning of ranking.

Furthermore, besides success/failure counts, populations of error reports (as created by $\mathcal{G}$) may have correlated characteristics that z-ranking will not take into account. One example is the free checker discussed in Section 5. With the free checker, there are some functions associated with a low number of success/failure counts that always free their arguments and do so by passing the freed argument to another function. The called function, however, may correspond to a highly ranked population of reports. The characteristics of the two report populations may be correlated, and the high ranking of one population should boost the ranking of the other. Extensions to the z-ranking methodology may possibly allow the ranking scheme itself to take such correlations into account.

9 Conclusion

This paper has explored and developed the idea of z-ranking, which uses frequency counts of successful and failed checks to rank error messages from most to least probable. We applied it to three different error checkers, two in-depth, and the last briefly. In practice it worked well: (1) true errors generally were pushed to the top of the ranking while (2) false positives were pushed to the bottom. Furthermore, application of the no-success heuristic often reduced the number of reports inspected substantially while still providing for all real bugs to be discovered; in one case roughly only a quarter of all reports were inspected. When compared to 1.0×10^5 randomized error rankings, z-ranking often scored in the top 1%. Moreover, within the first 10% of error report inspections, z-ranking found 3-7 times more bugs than the average number of bugs found by random ranking for the checkers we analyzed.

Furthermore, z-ranking made formerly unusable checkers effective. A good example was that the lock checker could not previously handle semaphores since they had two conflated uses: (1) as paired locks and (2) as unpaired atomic counters (each occurrence of which would generate a false message). Because our checker could not distinguish these cases, previously we had given up. With z-ranking we could easily find such errors.

We believe z-ranking would be useful in many static error checking tools: all tools must make analysis approximations and, as a result, they all have false positives. Z-ranking provides a simple way to control the impact of such approximations.

10 Acknowledgements

We thank Priyank Garg, Rushabh Doshi, Yichen Xie, Junfeng Yang, Seth Hallem, and Tony Abell for their invaluable comments on clarity and content. This research was supported by DARPA contract MDA904-98-C-A933. Ted Kremenek received additional funding through a National Science Foundation Graduate Fellowship.

References

1. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN 2001 Workshop on Model Checking of Software. (2001)
2. Das, M., Lerner, S., Seigle, M.: Path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany (2002)
3. Evans, D., Guttag, J., Horning, J., Tan, Y.: Lclint: A tool for using specifications to check code. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering. (1994)
4. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: SIGPLAN Conference on Programming Language Design and Implementation. (2000) 219–232
5. Aiken, A., Faehndrich, M., Su, Z.: Detecting races in relay ladder logic programs. In: Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (1998)
6. Wagner, D., Foster, J., Brewer, E., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: 2000 NDSSC. (2000)
7. Foster, J., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. (2002)
8. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2000)
9. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. (2001)
10. Freedman, D., Pisani, R., Purves, R.: Statistics. Third edn. W.W. Norton (1998)
11. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: SIGPLAN Conference on Programming Language Design and Implementation. (2002)
12. Ross, S.M.: Probability Models. Sixth edn. Academic Press, London, UK (1997)
13. Santer, T.J., Duffy, D.E.: The Statistical Analysis of Discrete Data. Springer-Verlag (1989)