ExPert: Dynamic Analysis Based Fault Location via Execution Perturbations

Neelam Gupta and Rajiv Gupta

University of Arizona
Dept. of Computer Science
Tucson, AZ 85721
{ngupta,gupta}@cs.arizona.edu

Abstract

We are designing dynamic analysis techniques to identify executed program statements where a fault lies, i.e. the fault candidate set. To narrow the set of statements in the fault candidate set, automated dynamic analyses are being developed which consider not only a failed run of a program but also execution perturbations of the failed run. The goal of this work is to focus the users attention on a small subset of statements in the fault candidate set[1].

1. Introduction

Programming being a primarily human activity, errors creep into software inspite of the advances made in the areas of programming languages and software development processes. Typically a programmer becomes aware of the existence of bugs in a program when he/she observes that a program output deviates from the expected output. A standard debugging process consists of setting breakpoints, reexecuting the program with the error-inducing input, and examining the program state (e.g., variable values, call stack, etc.) to understand the cause of incorrect output being generated. During this process, the programmer must decide what part of the execution to explore to isolate the bug. This process of exploration is often tedious and time consuming.

Fault location has proved to be a very difficult problem and while variety of approaches are being explored by researchers, when used individually, these approaches have met with limited success. We believe that the reason for this limited success is that the types of faults and types of *dynamic indicators* that point to the location of these faults may vary widely. Moreover, to locate faults with any *degree of precision* is complicated by additional factors. First it may be necessary to consider multiple dynamic indicators. Second multiple indicators may not be discernible from a single program run. In absence of results from a large number of runs being available, to uncover additional runs of relevance, we may need to carry out a targeted search for such runs via *execution perturbations* of the original failed run. Design of execution perturbation strategies will play a critical role in determining their usefulness.

In this paper we describe automated dynamic analyses that focus the software developer's on a small subset of statements such that the root cause of a failed run can be found in these statements. The smaller the subset the more likely it is that the user will locate the cause of failure quickly. Therefore we have developed highly aggressive dynamic techniques to uncover evidence that can be used to prune the set of executed statements and produce a relevant subset of statements for the programmer to examine. The novelty of our research lies in the types of evidence that are considered and the highly aggressive means that will be employed to uncover them.

Types of Evidence. We begin by identifying conservatively large set of statements in the failed run such that the faulty code is definitely captured in this *fault candidate set* – in fact this set can be trivially constructed by including all statements that were executed during the failed program run. Next we prune the fault candidate set first using *negative evidence* and then using *positive evidence*. Negative evidence is uncovered in form of a value, and the statement execution instance that produced the value, such that the value is known to be directly or indirectly related to the failure. Starting from such a value, and traversing the dynamic dependence graph appropriately, a subset of executed statements potentially involved in the failure are found. Different sources of negative evidence produce different approximations of the potentially fault candidate set. By intersect-

[1]This research was funded by NSF grant CNS-0614707 under the CSR Program.

ing these different approximations a smaller fault candidate set can be found. Positive evidence is uncovered in form of a value, and the statement execution instance that produced the value, such that we are very highly certain that the value is correct. Therefore the statement instances that produce such values can be further pruned from the fault candidate set.

Execution Perturbations. Simply identifying potential sources of negative and positive evidence is not enough. We require automated dynamic techniques for uncovering this evidence during a failed run. We describe an aggressive strategy for uncovering evidence that is based on the idea of *execution perturbations* – state at a selected execution point is carefully perturbed during the failed run, the effect of this perturbation on the failed run is observed, and through this observation important evidence is uncovered. This is a powerful approach for exploring *what if* scenarios, i.e. we can observe how the program would have behaved if a some of the program state would have been different.

2. Negative Evidence: Coarse-Grained Pruning of Fault Candidate Set

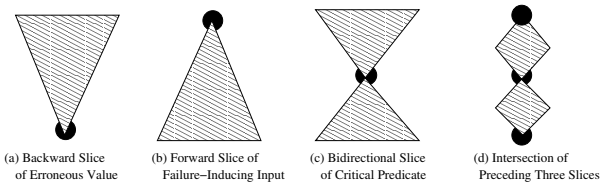


Figure 1. Types of Negative Evidence and Corresponding Dynamic Slices.

The most intuitive form of negative evidence that is immediately available to the programmer from a failed run is an erroneous output produced by the program – if the program crashes, the values used in the operation that caused the crash can be treated as *erroneous values*. By computing the *backward dynamic slice* of the erroneous value (which includes all executed statements that directly or indirectly influence the computation of the an erroneous value through a chain of dynamic data and/or control dependences) a fault candidate set can be identified. The backward dynamic slice of an erroneous value has been used as the basis of automated debugging for a long time [3]. Our work has identified two other forms of negative evidence. Although these two new forms are less intuitive, they can offer great assistance in fault location. First let us consider the concept of the *failure-inducing input difference*. Given a failed run, the failure-inducing input difference is the minimal part of the input which if changed causes the program to run successfully. In other words, the failure-inducing input difference plays a role in triggering the program failure and hence its *forward dynamic slice* (which includes executed statements that are directly or indirectly influenced by the failure-inducing input difference via a chain of data and/or control dependences) can also be considered as a fault candidate set [1]. Finally let us consider the notion of a *critical predicate* which is an execution instance of a conditional branch predicate in the failed run such that if the direction in which the branch goes is forced to switch, the program execution produces the correct output [8]. Switching the outcome of the conditional branch predicate can cause correct output to be produced under two situations: an error in the statements that influence the predicate outcome is undone by switching the outcome; or by switching the outcome we avoid execution of a statement which may be causing a program crash in the failed run. Thus, it makes sense to compute a *bidirectional dynamic slice* of the critical predicate which includes executed statements belonging to both the backward and forward dynamic slices of the critical predicate. The three types of slices based upon the three kinds of negative evidence are shown conceptually in Figure 1. Finally, the fourth and smallest fault candidate set shown in Figure 1 is computed by intersecting the fault candidate sets produced by the three different kinds of negative evidence. It should be noted that while set of all executed statements from the failed run form a fault candidate set, each of the dynamic slices shown in Figure 1 represent a pruning of this conservatively large fault candidate set.

We examined a set of reported bugs in real programs shown in Table 1 for the above types of negative evidence from which backward slice (BwS), forward slice (FwS), and bidirectional slice (BiS) is computed. Success in finding a particular type of evidence is shown by marking the entry with $\surd$ while failing to find evidence is indicated by $\times$. As we can see, it is important to look for different types of evidence because each type of evidence may not be uncovered in every case. For example, the failed runs of faulty versions of `grep` do not produce any output and hence backward slices could not be computed for them. For some failed runs of `flex` bidirectional slices could not be computed because correct output could not be produced by simply switching the outcome of a single branch predicate as the bug was more complex than what could be handled by switching the outcome of a single branch predicate.

In the remainder of this section we further expand upon the above ideas. We explain the challenges of uncovering the above forms of negative evidence and discuss the different forms of execution perturbations that are needed to effectively uncover the evidence.

Table 1. Uncovering Negative Evidence for some Faulty Versions.

Program	Bug Description	Source	↑ BwS	FwS ↓	↑ BiS ↓
flex 2.5.31	(a) some variable is not defined with option -l, which fails the compilation of xfree86	http://soureforge.net	√	√	√
	(b) string "]]" is not allowed in user's code	http://soureforge.net	√	√	×
	(c) the generated code contains extra #endif	http://soureforge.net	√	√	×
grep 2.5	using -i -o together produces wrong output	http://savannah.gnu.org	×	√	√
grep 2.5.1	(a) using -F -w together produces wrong output	http://savannah.gnu.org	×	√	√
	(b) using -o -n together produces wrong output	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/	×	√	
	(c) "echo dofe — grep dofe" finds no match	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/	×	√	√
make 3.80	(a) Backslashes in dependency names are not removed	http://savannah.gnu.org	√	√	√
	(b) Fail to recognize the updated file status while there are multiple target in the pattern rule	http://savannah.gnu.org	√	√	√
gzip-1.2.4	1024 byte long filename overflows into global variable	AccMon [13]	√	√	√
ncompress-4.2.4	1024 byte long filename corrupts stack return address	AccMon [13]	√	√	√
polymorph-0.4.0	2048 byte long filename corrupts stack return address	AccMon [13]	√	√	√
tar-1.13.25	wrong loop bounds lead to heap object overflow	AccMon [13]	√	√	√
bc-1.06	misuse of bounds variable corrupts heap objects	AccMon [13]	√	√	√
tidy-34132	memory corruption problem	AccMon [13]	√	√	√

2.1. Backward Slicing of Erroneous Value

A backward dynamic slice of a variable at a point in the execution includes all those executed statements which effect the value of the variable at that point. Dynamic slices identify a subset of *executed statements* that is expected to contain faulty code. The smaller the subset the better it is. Given an erroneous value, dynamic slicing identifies the subset of executed program statements that *influenced* the computation of the erroneous value.

Full slicing. Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data and/or control dependences* are included in full slices [3]. This is the most commonly used form of slicing and our work [9] has shown that it can often, but not always, capture faults in branch predicates as control dependences are also considered. Results of our studies reported in [10] show that the number of executed statements can range from 2.46 to 56.08 times the number of statements in a full slice.

Relevant slicing. While relevant slices also consider data and control dependences, in addition, they include statements such that omission of execution of those statements could have led to the computation of the faulty value [2]. Consider the following code fragment: $X = ..; \ if \ P \ then \ X = .. \ endif; \ Print(X)$. If fault is present in P which causes the predicate to evaluate to false instead of true, wrong value of X is output and more importantly the predicate P is missed by the full slice. Recognizing that omission of the execution of nested definition of X could be the cause of the error, P is identified as a relevant predicate which is also added to the dynamic slice. Our work has shown that if faults are present in predicates, they can quite often be missed by the full slices [9].

Identification of relevant predicates is a challenge as dynamic information collected only involves statements that were executed, not the statements whose execution was missed. It has been proposed that relevant predicates may be conservatively overestimated using static analysis [2]; however, overly conservative fault candidate sets are less useful to the programmer. We have addressed the challenge of identifying relevant predicates using a novel form of *execution perturbation* [6]. Essentially for a given conditional branch predicate, if the outcome takes us along the side of the branch that is definition-free wrt variable X, we would like to know if other side is not definition-free wrt X. We can dynamically determine this by intercepting the program at the appropriate instance of branch predicate and forcing the the program execution along the alternate side – if the other side is found not to be definition-free, the predicate is added to the relevant slice. This approach accurately finds relevant predicates and thus produces dynamic slices that capture the faulty code but are not unnecessarily large. Forcing the execution along an alternate path may result in a program crash. In this case either we can conservatively consider the predicate as relevant or we can suppress the execution of statements that cause a crash or use operands whose computation was suppressed to avoid a crash.

2.2. Forward Slice of Failure-Inducing Input

Zeller introduced the term *delta debugging* [11] for the process of determining the causes for program behavior by looking at the differences (the deltas) between the old and new configurations of the programs. Hildebrandt and Zeller [12] then applied the delta debugging approach to simplify and isolate the failure-inducing input difference. The basic idea of delta debugging is as follows. Given two program runs r_s and r_f corresponding to the inputs I_s and I_f respectively, such that the program fails in run r_f and completes execution successfully in run r_s, the delta debugging algorithm can be used to systematically produce a pair of inputs I'_s and I'_f with minimal difference such that program fails for I'_f and executes successfully for I'_s. The difference between these two inputs isolates the failure-inducing difference part of the input. These inputs are *critical inputs* whose values play a critical role in distinguishing a success run from a failing run. As already mentioned, we can use the minimal *failure-inducing input difference* for computing the forward dynamic slice. The data in Table 1 already indicates that this approach is effective. To further illustrate why this is the case we consider the following example.

We applied the above approach to a well known buffer overflow problem in `gzip-1.0.7`. Figure 2 illustrates the details of the problem. On the left hand side of Figure 2, we show the relevant code segment for the problem. The problem happens in the `strcpy` statement at line 844. Variable `ifname` is a global array defined at line 198. The size of the array is defined as 1024. Before the `strcpy` statement at line 844, there is no check on the length of string `iname`. If the length of string `iname` is longer than 1024, then buffer overflows and the program crashes.

The memory layout of gzip program is shown on the right side of Figure 2. We can see from the figure that there is a global pointer `env` located in an address space above array `ifname`. The difference between `env` and `ifname` is 3604 bytes. If the length of string `iname` is larger than 3604, the value of `env` will be changed due to buffer overflow. When we look at function `do_exit` at line 1341, before the program quits, it tries to free the memory pointed to by `env`. If the value of `env` is an illegal memory address due to buffer overflow, it causes a segmentation fault at line 1344.

To test the gzip program, we picked two inputs: the first input is a file name `'aaaaa'`, which is a successful input, and the second input is a file name `'a <repeated 3610 times>'`, which is failure input because the length is larger than 3604. After applying *sddmin* algorithm on them, we have two new inputs: the new successful input is a file name `'a <repeated 3604 times>'` and the new failed input is a file name `'a <repeated 3605 times>'`. The failure-inducing input difference between

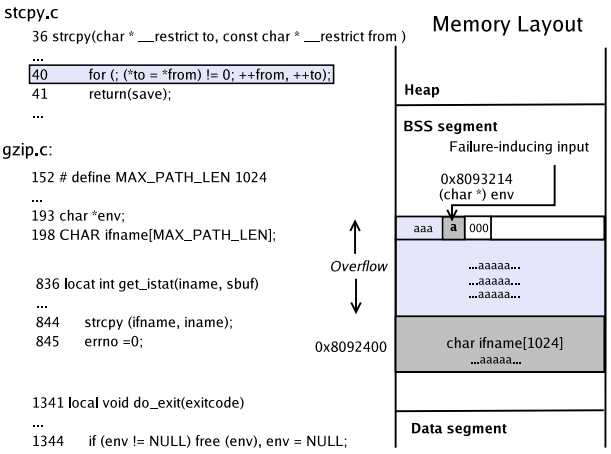


Figure 2. gzip Buffer Overflow.

them is the last character 'a' in the new failed input.

We used slicing to compute the forward slice of the failure-inducing input difference in the failed input and the backward slice of `env` at line 1344. The size of the forward slice is 4 which includes the `for` statement at line 40 in `strcpy.c`. This is exactly the place where the buffer overflow occurred. Our slicing implementations run on the binary code level and thus are able to check the memory space of a program and even check the code in the library.

The *execution perturbation* that is carried out for this technique is simply in the program input. An important issue that we need to explore is the selection of input pairs (one failing and one succeeding) from which to identify failure-inducing input difference. Different initial input pairs may isolate different failure-inducing input differences. In particular, if the program contains multiple faulty statements it may be desirable to consider a larger number of input pairs. We will develop algorithms that rank the statements belonging to failure-inducing forward slices of multiple influencing inputs to measure their likelihood for being faulty. Another critical issue related to identification of minimal failure-inducing input difference arises in presence of multiple errors in the program. When execution perturbations are performed the program execution may encounter another error and crash before reaching the program point at which an error was encountered in the original failed run. We will explore the use of *suppressing crashes* to ameliorate this problem.

2.3. Bidirectional Slice of Critical Predicate

Given an erroneous run of the program, the objective of this method is to perform execution perturbation that explicitly forces the control flow of the program along alternate branches of the predicates evaluated in the run so as

to identify critical predicates in the faulty run. The basic idea of this approach is inspired from the following observation. Given an input on which the execution of a program fails, a common approach to debugging is to run the program on this input again, interrupt the execution at certain points to make changes to the program state, and then see the impact of changes on continued execution. If we can discover the changes to program state that cause the program to terminate correctly, we obtain a good idea of the error that otherwise was causing the program to fail. However, automating the search of state changes is prohibitively expensive and difficult to realize because the search space of potential state changes is extremely large (e.g., even possible changes for the value of a single variable are enormous if the type of the variable is integer or float). On the other hand changing the outcomes of predicate instances greatly reduces the state search space since a branch predicate has only two possible outcomes, true or false. Therefore we note that through forced *switching* of the outcomes of some predicate instances at runtime, it may be possible to cause the program to produce correct results.

```
 970    base = ...
        . . .
2565    base[...] = ...
        . . .
2667    for ( i = 0; i <= lastdfa; ++i )
2668        {
            . . .
2673        int offset = base[i+1];
            . . .
2677        chk[offset] = EOB_POSITION;
            . . .
2681        chk[offset - 1] = ACTION_POSITION;
            . . .
2683        }
2684
2685    for ( i = 0; i <= tblend; ++i )
2686        {
            . . .
2690        else if ( chk[i] == ACTION_POSITION )
                printf("%7d, %5d,", 0 , ...);
            . . .
2696        else   /* verify, transition */
                printf("%7d, %5d," , chk[i], ...);
            . . .
2699        }
```

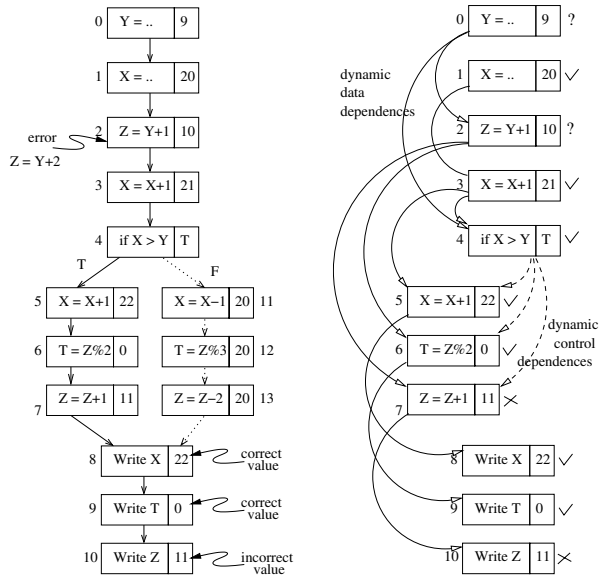
Figure 3. Example from Siemens suite.

We illustrate our idea with the faulty version of the $flex$ (a fast lexical analyzer generator) program shown in Figure 3 which is taken from the Seimens suite [14]. The Siemens suite provides the associated test suites and faulty version for each program. The program in Figure 3 is derived from $flex - 2.4.7$ and augmented by the provider of the program with a bug that is circled in the figure: $base[i + 1]$ should actually be $base[i]$. We took the first provided input which produced an erroneous output. We observed that the output is different from the expected output for the 538th character, a '1' is produced as output due to the execution of $printf$ in the $else$ part (line 2696) of the $else \quad if$ statement at line 2690 instead of a '0' that should be output by execution of the $printf$ in the $then$ part of the $else \quad if$ statement. Under the correct execution at line 2673 $offset$ would have been assigned the value of $base[0]$ which is 1. The variable $chk[0]$ at line 2681 would have been assigned ACTION_POSITION causing the predicate at line 2690 to evaluate to $true$ for loop iteration corresponding to $i = 0$. Due to the error at line 2673, an incorrect value of $offset(= 3)$ causes $ch[0]$ to have an incorrect stale value $(= 1)$ which causes the predicate at line 2690 to incorrectly evaluate to its $false$ outcome. Using our proposed method we looked for a predicate instance whose switching corrected the output. We found the appropriate instance of the $else \quad if$ predicate instance through this search. Once this predicate instance was found we could easily determine by following backwards the data dependences that the incorrect value of $ch[0]$ was a stale value and it did not come from most recent execution of for loop at line 2667. Thus, now it was clear that the error was in the statement at line 2673 which sets the $offset$ value. The above example also illustrates that it is important to alter the outcome of selected predicate instances as opposed to all execution instances of a given predicate. This is because the fault need not be in the predicate but elsewhere and thus all evaluations of the predicate need not be incorrect. In the $flex$ example we showed above, by enforcing the outcome of a predicate we avoided searching for potential modifications of values for $chk[\,]$, $offset$, or $base[\,]$ which are $integer$ variables and thus can take many different values.

The *execution perturbation* needed for locating critical predicates is similar to the one needed for the recovery of relevant predicates. So far we only discussed switching of a single predicate instance outcome. In general, more complex and interesting situations can exist for which more sophisticated changes in control flow are required. For example, the complexity of the fault may cause multiple instances of multiple distinct predicates to evaluate incorrectly. Thus in such a situation it will be necessary to alter several predicate outcomes during execution. Next, the program may contain multiple distinct faults and even though in the failing run a single fault is encountered, after switching a predicate outcome, additional faults may be encountered. Finally we will address issues of efficient implementation of multiple program executions required for this technique. We will explore tradeoffs between repeated full executions of the program to explore the search space and partial re-executions based upon an ability to rollback the execution to the desired point, forcing a change in a predicate's outcome, and continuing execution. We will develop an algorithm for automatically and dynamically selecting between these two options.

3. Positive Evidence: Fine-Grained Pruning

Next we describe a strategy for fine-grained pruning of the fault candidate set. For the purpose of this discussion let us assume that the fault candidate set is the backward dynamic slice. We observe that some of the statements used in computing the incorrect value may also have been involved in computing correct values (e.g., a value produced by a statement in the dynamic slice of the incorrect value may also have been used in computing a correct output value produced by the program prior to producing the incorrect value). Based upon further analysis of the program we may be able to determine that some of these statements are therefore very highly likely to be correct and thus they can be removed from the fault candidate set [7].



(a) Program Execution. (b) Dynamic Dep. Graph.

Figure 4. Pruning a dynamic slice.

Figure 4(a) shows an execution of program that follows the path corresponding to the true evaluation of the predicate at node 4. The value shown to the right of each statement is the value computed by the statement during execution. The dynamic dependence graph of this execution is shown in Figure 4(b) – the solid edges are data dependence edges while dotted edges are control dependence edges. The nodes in the dynamic slice of the incorrect output value produced by statement 10 include $\{0, 1, 2, 3, 4, 7, 10\}$. Now let us see how the correct outputs produced by statements 8 and 9 are used to mark the nodes in the backward full dynamic slice of the incorrect output as being correct or potentially faulty.

From the correct output value of X written by statement 8 we infer that the values produced by statements 1, 3 and 5 are also correct. The reasoning on which this inference is based is as follows. The statements 3 and 5 represent *one-to-one mappings* between the used operand values and generated result values of X. Therefore any change in the values produced by 1, 3 or 5 will cause the value output at statement 8 to change. However, the value output at statement 8 is known to be correct. Thus, we mark statements 1, 3 and 5 with $\surd$ indicating that they produce correct values. We further conclude that the *true* evaluation of predicate $X > Y$ is also correct. This is because if $X > Y$ would have evaluated to false, it would have produced a different output value for X at statement 8. Given the above observations, the pruned dynamic slice of incorrect value output at statement 10 will never include statements 1, 3, 4 and 5.

While in this paper we have briefly described our key ideas, more detailed experimental studies can be found in [5, 4, 7, 6].

References

[1] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *ASE*, 2005.

[2] T. Gyimothy, A. Beszedes, I. Forgacs, "An Efficient Relevant Slicing Method for Debugging", *ESEC/FSE*, 1999.

[3] B. Korel and J. Laski, "Dynamic Program Slicing," *IPL*, 29:3, 1988.

[4] X. Zhang, N. Gupta, and R. Gupta, "Locating Faulty Code By Multiple Points Slicing," *SP&E* journal, to appear.

[5] X. Zhang, N. Gupta, and R. Gupta, "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults," *Empirical Software Engineering* journal, to appear.

[6] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards Locating Execution Omission errors," submitted *PLDI* 2007.

[7] X. Zhang, N. Gupta, and R. Gupta, "Pruning Dynamic Slices With Confidence," *PLDI*, 2006.

[8] X. Zhang, R. Gupta, and N. Gupta, "Locating Faults Through Automated Predicate Switching," *ICSE*, 2006.

[9] X. Zhang, H. He, N. Gupta, and R. Gupta "Experimental Evaluation of Using Dynamic Slices for Fault Location", *AADEBUG*, 2005.

[10] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *PLDI*, 2004.

[11] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?," *ESEC/FSE*, 1999.

[12] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE TSE*, 28:2, Feb. 2002.

[13] P. Zhou et al., "Accmon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants," *MICRO*, 2004.

[14] http://www.cse.unl.edu/∼galileo/sir