PENUMBRA: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting

James Clause
College of Computing
Georgia Institute of Technology
clause@cc.gatech.edu

Alessandro Orso
College of Computing
Georgia Institute of Technology
orso@cc.gatech.edu

ABSTRACT

Most existing automated debugging techniques focus on reducing the amount of code to be inspected and tend to ignore an important component of software failures: the inputs that cause the failure to manifest. In this paper, we present a new technique based on dynamic tainting for automatically identifying subsets of a program's inputs that are relevant to a failure. The technique (1) marks program inputs when they enter the application, (2) tracks them as they propagate during execution, and (3) identifies, for an observed failure, the subset of inputs that are potentially relevant for debugging that failure. To investigate feasibility and usefulness of our technique, we created a prototype tool, PENUMBRA, and used it to evaluate our technique on several failures in real programs. Our results are promising, as they show that PENUMBRA can point developers to inputs that are actually relevant for investigating a failure and can be more practical than existing alternative approaches.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation, Reliability

Keywords

Failure-relevant inputs, automated debugging, dynamic information flow, dynamic tainting

1. INTRODUCTION

Debugging is known to be a labor-intensive, time-consuming task that can be responsible for a large portion of software development and maintenance costs [21, 23]. Common characteristics of modern software, such as increased configurability, larger code bases, and increased input sizes, introduce new challenges for debugging and exacerbate existing problems. In response, researchers have proposed many semi- and fully-automated techniques that attempt to reduce the cost of debugging (*e.g.,* [8, 9, 11–13, 18, 24, 25, 27]). The majority of these techniques are code-centric in that they focus exclusively on one aspect of debugging—trying to identify the faulty statements responsible for a failure.

Although code-centric approaches can work well in some cases (*e.g.,* for isolated faults that involve a single statement), they are often inadequate for more complex faults [4]. Faults of omission, for instance, where part of a specification has not been implemented, are notoriously problematic for debugging techniques that attempt to identify potentially faulty statements. The usefulness of code-centric techniques is also limited in the case of long-running programs and programs that process large amounts of information; failures in these types of programs are typically difficult to understand without considering the data involved in such failures.

To debug failures more effectively, it is necessary to provide developers with not only a relevant subset of statements, but also a relevant subset of inputs. There are only a few existing techniques that attempt to identify relevant inputs [3, 17, 25], with delta debugging [25] being the most known of these. Although delta debugging has been shown to be an effective technique for automatic debugging, it also has several drawbacks that may limit its usefulness in practice. In particular, it requires (1) multiple executions of the program being debugged, which can involve a long running time, and (2) complex oracles and setup, which can result in a large amount of manual effort [2].

In this paper, we present a novel debugging technique that addresses many of the limitations of existing approaches. Our technique can complement code-centric debugging techniques because it focuses on identifying program inputs that are likely to be relevant for a given failure. It also overcomes some of the drawbacks of delta debugging because it needs a single execution to identify failure-relevant inputs and requires minimal manual effort.

Given an observable faulty behavior and a set of *failure-inducing inputs* (*i.e.,* a set of inputs that cause such behavior), our technique automatically identifies *failure-relevant inputs* (*i.e.,* a subset of failure-inducing inputs that are actually relevant for investigating the faulty behavior). Our approach is based on dynamic tainting. Intuitively, the technique works by tracking the flow of inputs along data and control dependences at runtime. When a point of failure is reached, the tracked information is used to identify and present to developers the failure-relevant inputs. At this point, developers can use the identified inputs to investigate the failure at hand.

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

To evaluate our technique, we implemented it in a prototype tool, called PENUMBRA, and performed two empirical studies that assesses the feasibility and usefulness of our technique. Our first study analyzes PENUMBRA's performance in the context of debugging a set of real failures occurring in real applications. Our second study compares our technique against delta debugging. The results of our studies, albeit still preliminary, are promising. Our technique was able to (1) identify failure-relevant inputs that can be useful when debugging the considered failures and (2) produce results comparable to delta debugging but without needing complex setup or multiple executions.

The contributions of this paper are:

- A novel technique for identifying failure-relevant inputs that is based on dynamic tainting.
- A prototype tool, PENUMBRA, that implements our technique for x86 binaries.
- Two empirical studies that provide initial evidence of the feasibility and effectiveness of our technique.

The remainder of this paper is organized as follows: Section 2 discusses related work. Sections 3 and 4 provide background information and present a motivating example. Section 5 presents our technique. Section 6 describes our empirical evaluation and, finally, Section 7 presents our conclusions and sketches future research directions.

2. RELATED WORK

Because of the challenges and importance of debugging, there is large amount of work that is related to our technique. For the sake of space, we present only the most closely related of such approaches.

Given a failure and a set of failure-inducing inputs, delta debugging runs the failing program multiple times, each time with a different subsets of the inputs, in the attempt to find a minimized subset that still causes the program to fail [25]. As we mentioned in the Introduction, delta debugging is an effective technique, but its practical usefulness can be limited by the need for a precise oracle and by the time required to rerun a failing program many times. Mishergi and Su present an extension to delta debugging that can reduce the number of executions needed when operating on well-defined, tree-structured inputs [17], but currently no solution exists for the general case.

Chan and Lakhotia also propose several methods for identifying failure-relevant inputs [3]. The one most closely related to our technique uses dynamic slicing (described in the next paragraph) to trace incorrect elements in a program's output back to statements that read inputs that affect the incorrect output elements. Given these statements, developers must then identify which inputs are read by the statements. Compared to our technique, this approach requires much more manual interaction by developers, and as their evaluation shows, the amount of work involved becomes prohibitive on subjects as small as 150 lines of code.

The rest of the approaches that we discuss in this section are not alternative to our technique, but rather complementary—they are code-centric approaches that could be combined with our technique to increase the overall effectiveness of the debugging effort.

The first family of complementary techniques we discuss is based on slicing. Slicing is an analysis technique that uses

```
1. int x, y, z;
2. int a = input();
3. int b = input();

4. x = a - 2;
5. y = b * 4;
6. z = x + y;
```

(a)

```
1. int x, y;
2. int a = input();

3. if(a > 0)
4.    x = 0;
5. else
6.    x = 1;
7. y = 2;
```

(b)

Figure 1: Code examples that illustrate information flow through data and control dependencies.

data and control dependences in a program to locate a set of statements called a *slice*. The set of statements in a (backward) slice may influence the value of a variable at a particular program point. Weiser first proposed the use of *static slicing* for debugging [24]. Static slicing considers static data and control dependences for all possible executions and is often overly conservative. To address this problem, Korel and Laski proposed *dynamic slicing*, in which only dependences that were exercised during an execution are considered when computing slices [12]. More recently, Gupta and colleagues investigated the combination of delta debugging and dynamic slicing [8] to further reduce the size of slices, while Zhang and colleagues combined check-pointing with dynamic slicing for analyzing long-running applications [27].

A second family of complementary approaches attempts to automate debugging by ranking program entities. These approaches include techniques that aim to locate suspicious statements by comparing passing and failing executions. Agrawal and colleagues propose to use the set difference between statements executed in a failing run and statements executed in one or more passing runs as an indication of suspiciousness [1]. Renieris and Reiss improve on this approach by using a nearest neighbor calculation to find the passing execution that is most similar to the failing execution [19]. The closest passing execution is then used to calculate the set difference with the failing execution. Jones and colleagues propose a slightly different approach. Instead of using set difference, they estimate the suspiciousness of a statement by computing a ratio of its appearance in passing and failing executions [11]. Finally, Liblit and colleagues use a statistical approach that correlates predicates with failures [13].

3. BACKGROUND

In this section, we provide background information on dynamic tainting. Intuitively, dynamic tainting consists of (1) marking some data values in a program with a piece of metadata called a taint mark and (2) propagating taint marks according to how data flows in the program at runtime. In this way, dynamic tainting can track and check the flow of information through a program while it executes.

Information can flow through a program in two ways: through data dependences and through control dependences. We illustrate these two kinds of flows using the code examples in Figure 1. First, consider the code in Figure 1(a). Assume that variable a is tainted with taint mark t_a at line 2 and variable b is tainted with taint mark t_b at line 3. Given this assignment of taint marks, variables x, y, and z would be tainted, at the end of the execution, with sets of taint

marks $\{t_a\}$, $\{t_b\}$, and $\{t_a, t_b\}$, respectively. Taint mark t_a would be associated with x because the value of a is used to calculate the value of x (`x = a + 2`), that is, x is data dependent on a. Analogously, y would be tainted with t_b because the value of b is used to calculate the value of y (`y = b * 4`). Finally, z would be tainted with both t_a and t_b because the values of both x and y are used to compute the value of z (`z = x + y`), that is, z is indirectly data dependent on both a and b.

Consider now the code in Figure 1(b) and assume that variable a is tainted with taint mark t_a at line 2. Although a's value is not directly involved in the computation of x in this case, it nevertheless affects x's value: the outcome of the predicate at line 3 decides whether line 4 or line 6 will be executed, that is, the statements at lines 4 and 6 are control dependent on the statement at line 3. Therefore, the value of x at the end of the execution would be associated with taint mark t_a.

In the remainder of this paper, we refer to the propagation of taint marks along data dependences as *data-flow propagation*, the propagation of taint marks along control dependences as *control-flow propagation*, and the propagation along both data and control dependences as *data- and control-flow propagation*. A more thorough discussion of dynamic tainting can be found in the extensive literature on this topic (*e.g.,* [6, 15]).

4. MOTIVATING EXAMPLE

In this section, we provide an example that will be used in the remainder of the paper to illustrate our technique. Figure 2 shows the code for the example, which consists of a small program (`fileinfo`) for displaying information about one or more files. The command line arguments to the program are interpreted as an integer flag, which controls the verbosity of the program, and as a list of one or more file names. If the verbose flag is not set, the program prints the name and size of each file passed as an argument and the total size of these files. If the verbose flag is set, the program also prints the first fifty characters of each file.

Program `fileinfo` contains a memory-related fault: the call to `strcat` at line 16 can cause a heap overflow if (1) the verbose flag is set and (2) one of the files being processed has a size that requires more than nine digits to be represented numerically (*i.e.,* the file is one gigabyte or larger). The ten or more digits necessary for representing the file size, plus the fifty-five characters needed to display the preview (fifty for content and five for formatting and end-of-string character) overflow `out`, the sixty-four byte buffer allocated at line 8. Note that, because this failure is a heap overflow, its effect depends on the specific memory layout during a failing execution; it may cause a crash, produce a wrong result, or simply fail silently.

Although this program and fault are relatively simple, they possess several characteristics that make them a representative example of the type of debugging scenario that our technique targets. The first characteristic is the potential disparity between the amount of code and the input domain. There are less than thirty lines of code, but the amount of inputs (*i.e.,* the number and sizes of the files passed as parameters) can be extremely large. Second, understanding the failure requires tracing interactions among inputs coming from multiple sources. Finally, the number of failure-relevant inputs is likely to be a small percentage of all

```
   int main(int argc, char **argv) {
1.   int verbose, i, total_size = 0;
2.   struct stat buf;

3.   verbose = atoi(argv[1]);

4.   for(i = 2; i < argc; i++) {
5.     printf("%s: ", argv[i]);

6.     int fd = open(argv[i], O_RDONLY);
7.     fstat(fd, &buf);

8.     char *out = malloc(64);

9.     sprintf(out, "%lld", buf.st_size);

10.    if(verbose) {
11.      char *pview = calloc(55, 1);
12.      strcat(pview, " -[");
13.      int size = read(fd, pview+4, 50);
14.      pview[size+4] = '\0';
15.      strcat(pview, "]");

16.      strcat(out, pview);

17.      free(pview);
18.    }

19.    printf("%s\n", out);
20.    free(out);
21.    total_size += buf.st_size;
22.  }

23.  printf("total: %d\n", total_size);
   }
```

Figure 2: Code for the `fileinfo` utility.

inputs to the program; regardless of the sizes and number of files passed to `fileinfo`, there are only three failure-relevant inputs: the size and the first fifty characters of the first file larger than one gigabyte and `argv[1]` (the verbose flag).

5. OUR TECHNIQUE

In this section, we present our approach for automatically identifying failure-relevant program inputs. We first provide an intuitive description of the approach and then discuss its main characteristics in detail.

5.1 Overview

As we discussed in the Introduction, the overall goal of our technique is to help debugging by automatically identifying failure-relevant inputs. The basic intuition behind the approach is that dynamic tainting, due to its ability to mark and track data at runtime, can be successfully used to accomplish this goal. In this spirit, our approach works by (1) tainting each input read by the application with a unique taint mark, (2) tracking such inputs by suitably propagating taint marks during execution, and (3) identifying, when a failure occurs, the taint marks associated with data involved in the failure. The taint marks identified in the third step allow for separating failure-relevant inputs from inputs that are not relevant for debugging the considered failure.

Before presenting the technical details of our approach, we discuss how it would work on our `fileinfo` example from Figure 2. Consider an execution where the program is invoked as "`fileinfo 1 foo.txt bar.txt baz.txt`" and assume that the sizes of the three files passed as arguments are 512 bytes, 1024 bytes, and 1.5 gigabytes, respectively. This execution results in a failure because the verbose flag is set and the size of `baz.txt` is greater than one gigabyte.

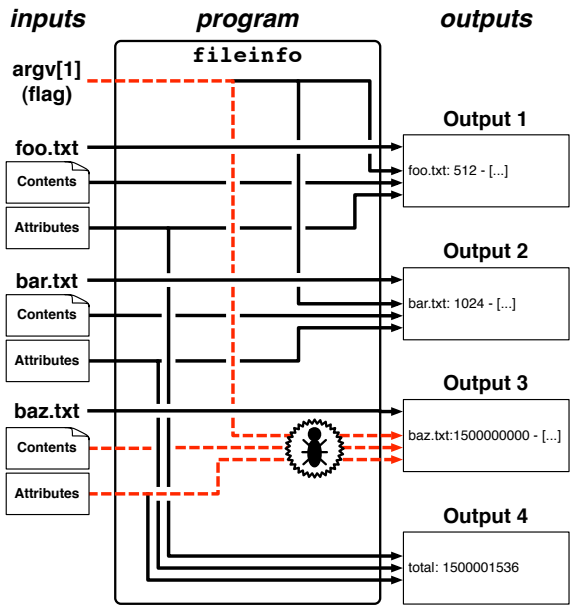

Figure 3: Intuitive view of the application of our technique to an execution of `fileinfo` from Figure 2.

Figure 3 provides an intuitive view of how our technique operates in this case. The left-hand side of the figure depicts the inputs to the program (*i.e.,* the command line arguments and the contents and attributes of the files provided as arguments). The name, content, and attributes of each file are shown separately because they are treated as distinct inputs by the program.

The right-hand side of the figure shows the outputs of the program. Because the verbose flag is set, and three files are passed as arguments, there are four outputs. The first three outputs are produced by the calls to `printf` at lines 5 and 19 and comprise the name, size, and first fifty characters (elided in the figure to conserve space) of the corresponding file. The fourth output, produced by the call to `printf` at line 23, shows the cumulative size of the three files. The bug icon in the figure represents the *relevant context* for the failure (*i.e.,* the point in the execution where the failure occurs and the data involved in the failure). For this example, the relevant context consists of the third execution of line 16 and the data in `pview` and `out`.

The lines that traverse the program illustrate, intuitively, how our technique tracks inputs at runtime, by associating unique taint marks to the inputs and propagating such marks as the inputs flow through the program. For example, the lines connecting the attributes of `foo.txt` to Outputs 1 and 4 indicate that there is a flow of information between these inputs and outputs in the execution considered. More specifically, the attributes affect the value of Output 1 because of a data dependence between `buf.st_size` (which is part of the attributes) and `out` at line 9; the attributes also affect the value of Output 4 because of a data dependence between `buf.st_size` (also part of the attributes) and `total_size` at line 23.

Our technique uses this information to identify which inputs are failure-relevant. When the failure occurs, the technique would determine which taint marks are associated with the data in the relevant context, identify the corresponding inputs, and report such inputs as failure-relevant. In our example, as Figure 3 intuitively depicts, the taint marks associated with the relevant context would map back to `argv[1]` (the verbose flag) and to contents and attributes of `baz.txt`. Therefore, our technique would report these inputs as failure-relevant to the developers.

In the remainder of this section, we discuss the main technical aspects of our technique by describing in detail its three parts: input tainting, taint propagation, and identification of failure-relevant inputs.

5.1.1 *Input Tainting*

Input tainting is responsible for associating taint marks with inputs as inputs enter an application. To do this, the technique intercepts all interactions between the application and all input sources (*e.g.,* keyboard, mouse, network connections, file systems) and suitably marks data coming from these sources.

Our technique assigns taint marks in one of three ways: per-byte, per-entity, or ad-hoc. The *per-byte* mode is the default tainting mode. As indicated by its name, this mode assigns a unique taint mark to each byte of input read by the application. By working at a low level of granularity, the per-byte mode has the potential to perform a fine-grained, and thus precise, identification of failure-relevant inputs. However, most inputs are structured, and applying a unique taint mark to each byte may be unnecessarily expensive in many cases. Consider, for instance, a call to function `gettimeofday`, which reads the system clock. Although multiple bytes are read by this function, they are unlikely to be meaningful individually, whereas together they represent the current time. By treating these bytes as a single input, the technique can reduce the number of taints marks that it needs without losing any precision. (Reducing the number of taint marks is beneficial because it may make the technique more scalable by speeding up the propagation of taint marks and reducing the amount of space needed to store taint marks.)

To address situations where per-byte tainting would be unnecessarily expensive, our technique provides a *per-entity* tainting mode. When possible, this mode treats related bytes as a single input and taints them using a single taint mark. There are many cases where relationships among individual bytes can be inferred. The bytes that compose individual command line arguments (*e.g.,* `argv[1][0...n]`, `argv[2][0...m]`) are likely to represent an atomic piece of information. Analogously, the bytes read by some common library functions (*e.g.,* `stat` or `fstat`), have a well-defined structure that can be leveraged when applying taint marks to them. For such cases, we created specific tainting strategies that apply a single taint mark to groups of related bytes.

The third option provided by our technique, the *ad-hoc* mode, allows developers to define custom taint assignment strategies. This mode is useful when developers' domain knowledge allows them to identify relationships among inputs that would not otherwise be inferred. Consider, for example, a program that reads structured records from a file. Without additional information, our technique would have no way of knowing the relationships among the contents of the file. A developer familiar with the system, however, could easily define a tainting policy that encodes this information and optimize the use of taint marks for such inputs.

Our technique performs two additional actions when assigning a taint mark t to an input i. First, it logs a pair

(t, src), where src is the source of the input. The source is expressed differently depending on its type (*e.g.,* a fully qualified path name for a file or a network address and port for a network connection). Second, it logs a pair (t, v_i), where v_i is the value of input i. Logging the values of inputs is necessary to handle cases where the input source is transient (*e.g.,* a network connection) and data read from that source is difficult, if not impossible, to recover. The information recorded by these actions is used by our technique when identifying failure-relevant inputs, as described in Section 5.1.3.

As a concrete example of how the tainting part of our technique operates, consider again the code in Figure 2. In this code, there are three locations where inputs are read by the application: variables `argc` and `argv`, the call to `fstat` at line 7, and the call to `read` at line 13. Our technique taints `argc` and `argv` in per-entity mode: the four bytes that compose `argc` are tainted with a single taint mark and all of the bytes in each of the arrays in `argv` are assigned the same taint mark (*e.g.,* `argv[0][0..n]` is assigned t_i, `argv[1][0..m]` is assigned t_j, etc.). The technique would also operate in per-entity mode for inputs read by `fstat`; by leveraging its knowledge about the structure of the data read by `fstat`, the technique would be able to assign a unique taint mark to each distinct part of the data (*e.g.,* file size, file owner, last access time). Finally, inputs read by the `read` function would be tainted in per-byte mode, with each of the bytes being assigned a unique taint mark.

5.1.2 Taint Propagation

As we mentioned in Section 3, a taint propagation policy specifies how taint marks are propagated during execution. Typically it is defined along two dimensions: how to combine taint marks and which types of dependences to consider. This section discusses our technique's propagation policy along these dimensions.

Our policy for combining taint marks is fairly intuitive. The technique taints all values written by a statement with the union of all taint marks associated with values read by that statement. For instance, after the execution of statement `x = y + z`, where `y` and `z` are tainted with taint marks t_1 and t_2, respectively, `x` would be associated with the set of taint marks $\{t_1, t_2\}$.

When choosing which dependences to consider, the two options are to propagate along only data dependences (*i.e.,* data-flow propagation) or to propagate along both data and control dependences (*i.e.,* data- and control-flow propagation).[1] To assess which option would be appropriate for our technique, we created several small code examples and applied our technique to them. During this preliminary investigation, we were able to construct both cases where data-flow propagation provided better results and cases where data- and control-flow propagation performed better.

Figure 4 shows, intuitively, the relationship between inputs identified using data-flow propagation and inputs identified using data- and control-flow propagation. The outermost circle represents the set of failure-inducing inputs (*i.e.,* the set of inputs that make the program fail). The union of the two gray circles represent the failure-relevant inputs (*i.e.,* the subset of failure-inducing inputs that are actually relevant for investigating the failure), which is our target. When using data- and control-flow propagation, our technique is conservative and identifies all inputs that may *affect* the data involved in the failure. As Figure 4 illustrates, the set of affecting inputs is a super set of the failure-relevant inputs. Therefore, there may be cases in which the set of inputs identified using data- and control-flow propagation may be too large to be useful to developers. Using data-flow propagation alone, our technique would identify smaller-sized sets, but at the cost of incompleteness. For subjects and faults such that the size of the darker grey set is negligible with respect to the size of the lighter grey set, data-flow propagation is likely to be preferable. Conversely, for subjects and faults where the size of the dark grey set dominates the size of the light grey one, data- and control-flow propagation should be used.

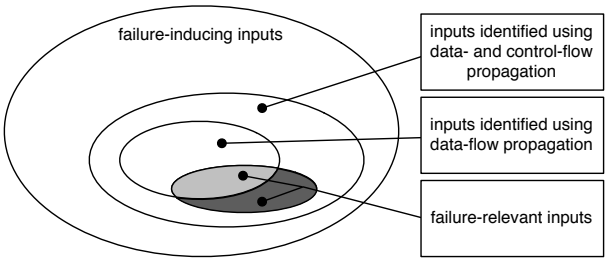

Figure 4: Impact of propagation choice when identifying failure-relevant inputs.

In general, there is no way to know a priori which propagation option will perform better for a specific program and failure. Therefore, we defined our technique so that it performs both kinds of propagation. The suggested usage scenario for the approach is one where developers start investigating a failure using data-flow propagation first and then, if necessary, use data- and control-flow propagation. As the results of our empirical study in Section 6.4 show, this usage scenario is effective for the subjects and faults we considered, and data-flow propagation alone tends to be more useful than data- and control-flow propagation.

5.1.3 Identification of Relevant Inputs

The third part of our technique is responsible for identifying which inputs are failure-relevant. The technique accomplishes this goal by (1) identifying the set of taint marks associated with the data portion of a relevant context and (2) mapping the identified taint marks back to inputs.

In a traditional manual debugging scenario, the starting point of the debugging process is what we defined as the relevant context: information on where the failure occurs and on which data is involved in the failure. Our technique requires exactly the same kind of information, encoded in the form of a developer-provided checking function. A *checking function* has two responsibilities. It must identify when the target failure occurs and it must report which data are involved in the failure. In the `fileinfo` example in Figure 2, for instance, a call to a checking function could be inserted before line 16. When the failure occurs, the function would inform the technique that the contents of `pview` and `out` are the data involved in the failure. Additional details about how checking functions are constructed in practice can be found in Section 6.3.

Given the relevant context for a specific failure, our technique identifies, when such failure occurs, the taint marks associated with the data involved. The identified taint marks

[1]Performing control flow propagation alone is not an option, as dependence chains must also involve data dependences.

are then mapped to the sources and values of the corresponding inputs. This information is extracted from the logs generated during input tainting (*cf.* Section 5.1.1). Inputs identified as failure-relevant are then provided to developers, who can use them to study the cause of the failure under investigation.

6. EVALUATION

Like most debugging aids, a completely realistic evaluation of our technique would require a comprehensive, full-fledged user study. However, at this stage of our research, we not not believe that the costs associated with this type of human study are justified, and we are mainly interested in gathering initial evidence of the usefulness of the approach. To this end, we implemented our technique in a prototype tool, PENUMBRA, and performed two case studies. In the first study, we investigate whether the information provided by PENUMBRA can effectively assist developers in debugging real failures. In the second study, we perform a comparison between our technique and delta debugging along two dimensions: the amount of manual effort needed to use the tools and (an estimate of) the time needed to diagnose and fix a considered failure given the results provided by each tool. The following sections discuss PENUMBRA, our subjects, and our experimental protocol and results.

6.1 Prototype Tool

PENUMBRA is a prototype tool that implements our technique. The current implementation consists of two components: a trace generator and a trace processor.

The trace generator is built on top of DYTAN, a generic dynamic tainting framework for x86 binaries that we developed in previous work [6]. The trace generator leverages DYTAN's built-in functionality for assigning and propagating taint marks to create a trace file that is then processed by the trace processor. The trace generator also supports the summarization of functions by using a short description of how taint marks propagate from the inputs to the outputs of such functions. For example, instead of recording how taint marks propagate along every statement executed inside of the `strcpy` function, the trace generator only needs to record that the outputs of the function should be assigned the same taint marks that are assigned to the inputs. Summarization can greatly reduce the size of the traces that are generated and also reduces the amount of time needed to process the traces. Currently, the trace generator summarizes most of the commonly used functions in the C library. In the future we plan on allowing users to add custom summarizations for additional functions.

The trace processor is implemented in Java. It takes as input trace files generated by the trace generator and computes failure-relevant inputs using data-flow propagation and data- and control-flow propagation.

Operating at the binary level has several advantages (*e.g.,* it allows PENUMBRA to handle shared libraries), but it also introduces several challenges. The most serious challenge is the difficulty of obtaining precise control-flow information. For example, in the presence of indirect jumps, it may be impossible to construct an accurate control-flow graph for a binary program. This imprecision can, in turn, negatively impact the accuracy of taint mark propagation; with an inaccurate control-flow graph, PENUMBRA must be conservative when identifying control-dependent regions and, consequently, may over-propagate taint marks. To reduce the impact of this kind of imprecision, we implemented a static analysis proposed by McCamant and Ernst that can help guide propagation along actual, rather than spurious, control dependences [16].

Currently, PENUMBRA ignores GUI inputs because handling them would require a considerable amount of additional implementation effort, while their omission does not prevent us from suitably evaluating the technique.

Table 1: Subject applications.

Application	*KLoC*	*Fault location*
bc-1.06	10.5	more_arrays : 177
gzip-1.2.4	6.3	get_istat : 828
ncompress-4.2.4	1.4	comprexx : 896
pine-4.44	239.1	rfc822_cat : 260
squid-2.3	69.9	ftpBuildTitleUrl : 1024

6.2 Subjects

The ultimate goal of our technique is to direct a developer's attention to a subset of inputs that are useful for debugging a failure. To perform a meaningful evaluation of our technique, we must therefore consider subjects that process a large amount of inputs. (If we considered programs with few inputs, it would be difficult to assess the benefit that our technique can provide.) With this goal in mind, we selected from related work [5, 27] and from the BugBench benchmark suite [14] five applications: `bc` version 1.06, an interpreter for a numeric processing language; `gzip` version 1.2.4 and `ncompress` version 4.2.4, two file compression utilities; `pine` version 4.44, an email and news client; and `squid` version 2.3, a caching proxy. These programs contain real faults (one per subject) and read many inputs from multiple sources. Table 1 shows, for each of the five subjects, its name and version (*Application*), its size in thousands of lines of code (*KLoC*), and the location of the fault considered in terms of containing function and line number (*Fault location).*

For each subject, we also selected a set of failure-inducing inputs: for `bc` we used the inputs provided with the subject; for `gzip`, `ncompress`, and `squid`, we constructed an appropriate set of inputs using information provided by the authors of BugBench; and for `pine` we used information provided by SecurityFocus.[2] We present additional details on the set of failure-inducing inputs for each subject in Section 6.4.

6.3 Experimental Protocol and Results

To collect the experimental data for PENUMBRA, we performed the following steps for each subject and recorded the wall-clock time each step took to complete:

Setup: To setup PENUMBRA, we ran the subject with its corresponding set of failure-inducing inputs and used traditional debugging techniques to manually identify the failure's relevant context. To avoid biasing the results, we selected the line of code where the considered failure occurs—identified by inspecting the crash dump generated by the failure—and all data read on that line. In general, developers are free to modify

[2] `http://www.securityfocus.com/bid/6120/exploit`

Table 2: Experimental data for investigating Study 1 and Study 2.

Application	*#Inputs*	**Penumbra**						**Delta Debugging**			
		# Relevant Inputs		*Time (s)*				*# Remaining*	*Time (s)*		
		DF	DF & CF	Setup	Exec.	Post.	Total	*Inputs*	Setup	Exec.	Total
gzip-1.24	10,000,056	1	3	163	295	9,307	9,765	1	1,800	129	1,929
ncompress-4.2.4	10,000,056	1	3	70	185	6,053	6,308	1	1,800	137	1,937
pine-4.44	15,103,766	26	15,100,344	97	80	2,117	2,294	90	5,400	7,036	12,436
bc-1.06	1373	209	743	314	4	4	322	285	12,600	1,727	14,327
squid-2.3	1,402,056	89	2,056	125	137	212	443	—	—	—	—

the relevant context based on any additional information, intuition, or domain knowledge that they may have regarding the failure being debugged. Note that, with some additional effort, it may be possible to completely automate this step, at least in some cases, by using tools such as Valgrind's MemCheck [20]. In fact, Tucek and colleagues have demonstrated the feasibility of this idea for certain types of failures [22].

Execution: In the execution step, we ran the subject with its corresponding set of failure-inducing inputs and used the trace generation component of PENUMBRA to generate a trace file. To avoid bias, we did not augment the trace generator with any ad-hoc taint assignment strategies (*cf.* Section 5.1.1) or custom summarizations (*cf.* Section 6.1).

Post processing: Finally, in the post processing step we ran the generated trace file through the trace processing component of PENUMBRA to identify failure-relevant inputs, considering both data-flow propagation and data- and control-flow propagation.

To collect the experimental data for delta debugging we used a similar procedure; we performed the following steps for each subject and recorded the wall clock time that was needed to complete each step.

Setup: Delta debugging's setup involves the construction of an automated oracle that is capable of reliably detecting the failure at hand. Note that creating an automated oracle for delta debugging is considerably more complex than constructing a checking function for PENUMBRA. First of all, the oracle must be robust and general; unlike a checking function, it must be able to identify the failure considered not only for the specific failing execution, but also for all the executions performed by delta debugging on subsets of the failure-inducing inputs. Moreover, because the failures we are considering cause crashes, but do not produce incorrect output, the oracle must take into account the internal state of the subject as the subject executes. We built such oracles by using `gdb` to inspect the stack trace at the failure point and the values of several, manually identified, pieces of program data.

In addition to observing the internal state of the applications, oracles for delta debugging also need to consider how long an execution should take to complete. Because delta debugging removes inputs, there are cases where an execution may not terminate. This is typically the case for interactive applications, such as `bc` and `pine`. To handle this situation, oracles typically use a timeout value and terminate executions that continue for longer than such timeout. For executions that are terminated because of the timeout, the oracle assumes, possibly incorrectly, that inputs used for the execution did not cause the failure. And if the oracle's assumption is incorrect (*i.e.,* the execution would have failed if it were allowed to continue executing), delta debugging will produce incorrect results. In contrast, our technique does not need to be concerned with execution times since it only requires a single run of the originally-observed failing execution to identify failure-relevant inputs.

Because a timeout that is too short may cause incorrect results, it is important to err of the side of longer values when selecting timeouts. For `bc` and `pine` we found that a 1-second timeout was long enough to prevent incorrect results.

Execution: In delta debugging's execution step we generated a subset of the subject's failure-inducing inputs by running an implementation of delta debugging provided by Zeller[3] on the subject with the subject's corresponding set of failure-inducing inputs and the automated oracle created in the previous step as inputs.

Table 2 presents the experimental data that we generated using the steps described above. The table is divided by vertical double bars into three sections. The left-most section shows the name and version of each subject (*Application*) and the number of inputs in the subject's set of failure-inducing inputs (*# Inputs*). Note that the number of inputs refers to the number of times PENUMBRA assigned a taint mark to an input. In other words, inputs that are not actually read by the application are not counted. For example, if an application took a ten megabyte file as input but only read the first five bytes of the file, we would consider the number of inputs to be five (rather than ten million). Counting inputs in this way eliminates the risk of obtaining artificially inflated results in our favor.

The middle section of the table shows the experimental data collected for PENUMBRA. The first two columns in this section show the number of failure-relevant inputs identified using data-flow propagation (*DF*) and data- and control-flow propagation (*DF & CF*). The final four columns show, in seconds, the wall-clock time measurements for each of the three steps in the experimental protocol for PENUMBRA described above (*Setup*, *Exec.*, and *Post.*) and the sum of these measurements (*Total*).

The right-most section of the table shows the experimental data for delta debugging. The first column in this section shows the number of inputs included in the minimized subset of failure-inducing inputs (*# Remaining Inputs*) and the

[3] `http://www.st.cs.uni-saarland.de/dd/DD.py`

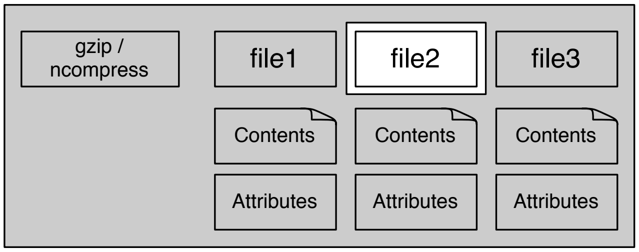

Figure 5: Failure-relevant input for `gzip` and `ncompress` identified using data-flow propagation.

next three columns show, in seconds, the wall-clock time measurements for the two steps in delta debugging's experimental protocol (*Setup* and *Exec.*) and the sum of these measurements (*Total*). Note that the time measurements for the setup step were recorded in half hour intervals. The dashed lines in the last row indicate that, after a day of work, we were unable to construct an automated oracle for the failure in `squid`. Constructing an oracle for the failure in `squid` is particularly difficult because `squid` is multithreaded, the failure involves interactions between multiple processes, and a considerable amount of infrastructure is necessary for controlling the environment in which `squid` executes (*e.g.,* redirecting `squid`'s network accesses to the minimized versions of web pages and data generated by delta debugging).

Threats to validity. The fact that we performed the steps necessary for generating this data ourselves is clearly a threat to the external validity of the experiments, and the conclusions that we draw may not generalize. These threats are mitigated by the fact that we are equally skilled with both techniques, as we used delta debugging extensively in previous work [7]. Therefore, even if the amount of time necessary for each step changes when they are performed by other developers, it is likely that the ratio between them will remain fairly constant. Another threat to the external validity of the study is that we used a limited number of subjects and faults. However, both the subjects and the faults we considered are real, and the debugging scenario investigated is representative of a real scenario.

6.4 Study 1: Effectiveness for Debugging

The goal of our first empirical study is to investigate whether the information provided by PENUMBRA is effective in assisting developers when debugging real failures. For each subject, we discuss the information generated by PENUMBRA and provide a qualitative assessment of its usefulness to developers. In the discussion, we assume that developers are following the scenario we presented in Section 5.1.2: first they consider failure-relevant inputs identified using data-flow propagation and then, if necessary, consider failure-relevant inputs identified using data- and control-flow propagation.

Gzip and Ncompress. We present these two subjects together because they are very similar. They are both file compression utilities and their faults are nearly identical: a buffer overflow occurs if a file with a path length greater than 1024 characters is compressed. Therefore, we use the same set of failure-inducing inputs for both subjects.

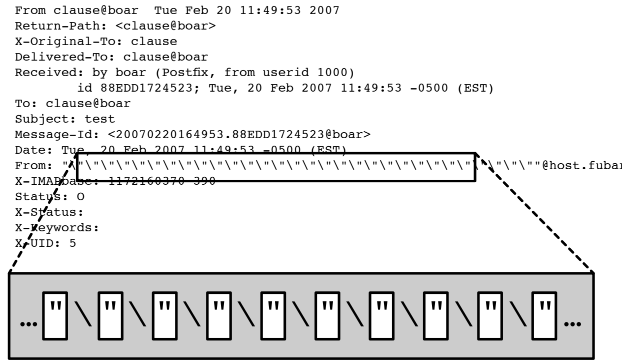

Figure 6: Excerpt of failure-relevant input for `pine` identified using data-flow propagation.

Figure 5 shows a representation of the inputs used to reveal the faults in `gzip` and `ncompress`. Each program is given three files to compress: `file1`, `file2`, and `file3`, where `file2` has a path length greater than 1024. As shown in Table 2, the total number of failure-inducing inputs is 10,000,056. The majority of these inputs (10,000,000) are read from the contents of `file1`. The remainder comes from several sources (*e.g.,* `argc`, `argv`, and `file1`'s attributes). Note that the contents and attributes of `file2` and `file3` are not counted as failure-inducing inputs because they are not read before the applications fail.

Using data-flow propagation, PENUMBRA identifies `file2`'s path as the only failure-relevant input. In Figure 5, the single failure-relevant input is shown inside of a white box. This is an encouraging result because the information provided by PENUMBRA correctly suggests two conclusions: (1) that `file2`'s path is involved in the failure and (2) that the rest of the failure-inducing inputs are not involved in the failure. These conclusions closely match the actual failure conditions and are likely to direct developers right at the core of the problem.

For this failure, using data- and control-flow propagation would lead to only a minimal increase in the number of inputs identified as failure-relevant. Only two additional inputs would be identified and reported by PENUMBRA: the name of the executable and the path of `file1`. The contents of `file1` would still not be reported, and developers would still be presented with information that is likely to point them directly to the problem.

Pine. The failure in `pine` is triggered when the program processes a mailbox containing an email with a "from" field that contains 22 or more double quotes. The failure-inducing inputs we selected include a mailbox with a size of approximately 15 megabytes that contains an email with a suitably-crafted "from" field. Figure 6 shows an excerpt of such problematic email.

Using data-flow propagation, PENUMBRA identifies 26 inputs as failure-relevant: the 26 double quotes in the problematic "from" field. The zoomed view in Figure 6 shows these inputs inside white boxes. This is also an encouraging result. Like in the cases of `gzip` and `ncompress`, the information provided by PENUMBRA using data-flow propagation correctly highlights a subset of the failure-inducing inputs that is necessary for the failure to manifest.

```
/* An example that finds all primes between 2 and limit. */

define tests (limit) {

    auto a1, a2, a3, ..., a62, a63, a64

    p1[1]   =  1;
    p2[1]   =  1;
    p3[1]   =  1;
    p4[1]   =  1;
    p5[1]   =  1;
    ...
    p29[1]  =  1;
    p30[1]  =  1;
    p31[1]  =  1;
    p32[1]  =  1;
    p33[1]  =  1;
    p34[1]  =  1;
    ...
    p68[1]  =  1;
    p69[1]  =  1;
}

print "\ntyping 'tests (10)' will construct 10 elements array.\n"
quit
```

Figure 7: Excerpt of failure-relevant input for `bc` identified using data-flow propagation.

Differently from the failures in `gzip` and `ncompress`, for this failure, data- and control-flow propagation identifies many additional inputs, nearly 15 million—almost the whole mailbox. This is an example of a situation where a large number of inputs potentially affect the data involved in a failure, but relatively few are actually relevant for investigating the failure. On the positive side, the relevant inputs identified through data-flow propagation for `pine` are likely to be sufficient for diagnosing the failure, so the developer would not need to consider this second set of inputs.

Bc. The fault in `bc` manifests when the program processes a script that (1) allocates at least 32 arrays and (2) has allocated more variables than arrays when the 32^{nd} array is created. Figure 7 shows the relevant portions of the problematic script. In the language that `bc` processes, variables can be allocated using the `auto` keyword (*i.e.,* `auto a1, a2`, etc.), and arrays are allocated when they are referenced (*i.e.,* `p1[1], p2[1]`, etc.).

Using data-flow propagation, PENUMBRA identifies 209 inputs as failure-relevant. These are the inputs shown inside the white boxes in Figure 7 and correspond to the names of variables and arrays in the script. The set of failure-relevant inputs only contains the names of 31 arrays, which is one short of the 32 arrays that are needed to trigger the failure. However, the identified inputs strongly suggest that the considered failure involves interactions between the processing of variables and arrays, so we believe this result is still promising.

Using data- and control-flow propagation for this failure causes an additional 534 inputs (743 total) to be identified as failure relevant. Although this is still a subset of the total inputs, the large number of inputs identified as failure-relevant is likely to obscure the actual cause of the failure.

Squid. The fault in `squid` manifests when the program processes an FTP request where the username, password, and host name components of the request combined contain more than 64 characters, which rfc1738[4] defines as unsafe. The

[4] `http://www.w3.org/Addressing/rfc1738.txt`

```
...
GET http://www.cc.gatech.edu/~clause
GET http://www.cc.gatech.edu/~orso
GET http://www.google.com
GET http://www.cc.gatech.edu/~clause
GET ftp://username#####...#####:password#####...#####@127.0.0.1
GET http://www.cc.gatech.edu
GET http://www.espn.com
GET http://yahoo.com
...
```

Figure 8: Excerpt of failure-relevant input for `squid` identified using data-flow propagation.

failure-inducing inputs we selected include a sequence of 50 HTTP and FTP requests, drawn from the authors' daily internet usage, that fetch data and webpages totaling approximately 1.5 megabytes in size. The seqeunce of requests also contains a suitably crafted FTP request that causes the failure. Figure 8 shows an excerpt of the request sequence that includes the problematic FTP request, in addition to several other HTTP requests.

Using data-flow propagation, PENUMBRA identifies 89 inputs as failure-relevant. These inputs are shown in Figure 8 surrounded by white boxes. Like for the other failures we have investigated, PENUMBRA's performance using data-flow propagation is encouraging. The failure-relevant inputs suggest that the username, password, and host name components of the problematic FTP request are involved in the failure, whereas the webpages and data read from the network are not involved in the failure.

Using data- and control-flow propagation, an additional 1967 inputs are identified as failure-relevant (2056 total). These additional inputs correspond to the majority of the requests, but the contents of the webpages and FTP data are still completely excluded. Therefore, even when considering this larger subset, there would still be a significant reduction in the amount of data developers would need to examine when investigating the failure.

Overall Considerations. Based on the results of this first empirical study, we can make some initial observations about the effectiveness of the information provided by our technique. For all of the five subjects considered, PENUMBRA computed failure-relevant inputs that closely correspond to the necessary conditions for reproducing the considered failures. Moreover, for these failures, only data-flow propagation was needed to provide useful information. Additionally, for the considered failures in `gzip`, `ncompress`, and `squid`, data- and control-flow propagation also provided results that are likely to be useful to developers. Conversely, for `bc` and `pine`, data- and control-flow propagation seems to identify too many inputs for the information to be useful. However, because data-flow propagation provides useful information in all of the cases we examined, according to our scenario of usage, developers would be unlikely to consider the use of data- and control-flow propagation. Therefore, we believe that the results of this case study, albeit preliminary, suggest that our technique can be effective in assisting debugging of failures in real applications.

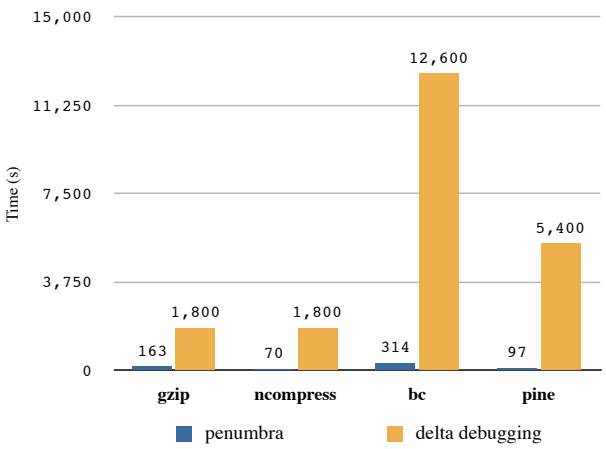

Figure 9: Comparison between the amount of setup time needed for penumbra and delta debugging.

6.5 Study 2: Comparison with Delta Debugging

The goal of our second study is to provide a quantitative comparison between PENUMBRA and delta debugging. In our experience, there are two main dimensions that developers consider when comparing debugging tools: (1) the amount of manual effort involved in using the tools and (2) the amount of time needed to diagnose and fix a considered failure given the information provided by the tools.

As a proxy for the amount of manual effort needed to use the tools, we used the amount of time needed to complete their setup step. For both tools, the other steps can be fully-automated and performed off-line (*e.g.,* overnight, or as part of an automated build system). Therefore, the time necessary for completing these steps is of little relevance, as long as it is of the same magnitude for both tools (which is the case for PENUMBRA and delta debugging).

Figure 9 shows, for `gzip`, `ncompress`, `bc`, and `pine`, a comparison between the amount of setup time needed to use PENUMBRA and the amount of setup time needed to use delta debugging. (These numbers are the same as the ones in Table 2 and are repeated here for the convenience of the reader.) As the figure shows, for these four subjects the amount of setup time needed for PENUMBRA is far less than the amount needed for delta debugging. This large difference in the amount of setup time between the two tools is likely to make PENUMBRA more practical and to be a strong incentive for its use by developers.

As an estimate of the relative amounts of time needed to diagnose and fix failures, given the information provided by the tools, we compared the number of inputs that each tool identifies as relevant.

Table 2 shows that, for the failures we considered, the number of inputs that PENUMBRA identifies using data-flow propagation is relatively close in size to the number of inputs identified by delta debugging. In addition, we found that the inputs identified by delta debugging are a super set of the inputs identified by PENUMBRA using data-flow propagation. If we consider the amount of relevant inputs for a failure to be an indicator of the amount of developers' time required to diagnose such failure, we expect that the two tools will have comparable performance, with PENUMBRA being slightly more efective, at least in some cases.

Table 2 also shows that for two subjects, `bc` and `pine`, the number of failure-relevant inputs identified by PENUMBRA using data- and control-flow propagation is significantly larger than the number of inputs identified by delta debugging. This means that for the failures in these subjects, the amount of time needed to diagnose and fix the considered failure using information provided by delta debugging will likely be less than the amount of time needed when using information provided by PENUMBRA using data- and control-flow propagation.

One final point to consider in this comparison is the reduction in the number of program statements that developers need to consider when investigating a failure that each tool provides. The set of minimized inputs that delta debugging provides corresponds to a minimized execution that still produces the considered failure and that can be readily used by the developer. Conversely, our technique identifies failure-relevant inputs, but does not automatically produce a minimized execution. Although it would be straightforward to compute a dynamic slice starting from such inputs (PENUMBRA itself could be easily modified to produce such a slice), computing an actual minimized execution would require some less straightforward extension of the technique.

Based on the results our second case study, we believe that PENUMBRA compares favorably with delta debugging and provides several incentives for its use in practice. First, for the subjects and failures we considered, PENUMBRA needed considerably less setup time than delta debugging, which suggests that PENUMBRA will also require less manual effort. Second, the number of failure-relevant inputs identified by PENUMBRA using data-flow propagation is comparable in size to the number of inputs identified using delta debugging. This observation, combined with the results from the first case study (which suggest that data-flow propagation may be sufficient to produce useful results in most cases), indicates that PENUMBRA and delta debugging are likely to provide a similar level of effectiveness for developers who are diagnosing and fixing failures.

Finally, we note that, although both tools share the same goal, they are not mutually exclusive. In practice, delta debugging could be used in conjunction with PENUMBRA by slightly extending the usage scenario we proposed in Section 5.1.2, as follows. Developers could first use PENUMBRA, which requires a smaller amount of setup time, to investigate a failure. If the information provided by PENUMBRA, using data-flow propagation first and data- and control-flow propagation later, is sufficient for diagnosing the failure, developers would terminate their debugging activity. Otherwise, they could invest the additional time needed to setup and run delta debugging and use the information it provides to further investigate the considered failure.

7. CONCLUSIONS AND FUTURE WORK

We presented a novel technique based on dynamic tainting for automatically identifying failure-relevant program inputs. Our technique is highly automated and complements existing code-centric debugging techniques by taking into account the second component of software failures: failure-inducing inputs. We also presented PENUMBRA, a prototype tool that implements our technique for x86 binaries and that we used to perform an empirical evaluation of our approach. Our results show that our approach is feasible and promising: for the subjects and failures we considered, PENUMBRA

was able to point developers to the parts of the inputs that were actually relevant for the failure. Our results also shows that our technique, by being highly automated, can overcome some of the practical limitations of existing techniques.

In addition to performing additional empirical evaluation with more subjects and real developers, we see several directions for future work. One immediate direction is to combine our technique with existing code-centric approaches, which can be done in several ways. First, we could use our approach to produce a dynamic slice from the identified failure-relevant inputs and feed the statements in the slice to a technique that ranks program statements, such as Tarantula [10]. Because such techniques compare statements in passing and failing executions, reducing the number of statements should reduce the noise in the data and result in more accurate rankings (as also shown in related work [26]). Second, we could use the information provided by our technique to assist in ranking entities. For example, a technique that ranks program statements could also consider the taint marks associated with the data accessed by each statement. Statements that access data with a particular taint could be considered more or less suspicious and ranked appropriately.

Acknowledgements

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

8. REFERENCES

[1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE '95: Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 143–151, 1995.

[2] M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. In *WODA '08: Proceedings of the 6th International Workshop on Dynamic Analysis*, 2008.

[3] T. W. Chan and A. Lakhotia. Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance*, 10(2):111–150, 1998.

[4] R. Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: A new approach to revealing neglected conditions in software. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 163–173, 2007.

[5] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 284–292, 2007.

[6] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[7] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 261–270, 2007.

[8] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2005.

[9] W. Ji, J. Xiao-xia, L. Chang, Y. Hai-yan, L. Chao, and J. Mao-zhong. A statistical model to locate faults at input level. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 274–277, 2004.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[12] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[14] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaulating bug detection tools. In *Bugs 2005: Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[15] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 198–209, 2004.

[16] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, 2008.

[17] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, 2006.

[18] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 5–15, 2007.

[19] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[20] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[21] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report 7007.011, National Institute of Standards and Technology, 2002.

[22] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 131–144, 2007.

[23] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A Process Analysis*, 23(5):459–494, 1985.

[24] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[25] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[26] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the 6th International symposium on Automated Analysis-driven Debugging*, pages 33–42, 2005.

[27] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 81–91, 2006.