Path Projection for User-Centered Static Analysis Tools

Khoo Yit Phang   Jeffrey S. Foster   Michael Hicks   Vibha Sazawal

University of Maryland, College Park

{khooyp,jfoster,mwh,vibha}@cs.umd.edu

Abstract

The research and industrial communities have made great strides in developing sophisticated defect detection tools based on static analysis. However, to date most of the work in this area has focused on developing novel static analysis *algorithms*, and neglected study of other aspects of static analysis *tools*, in particular user interfaces. In this work, we present a novel user interface toolkit called Path Projection that helps users visualize, navigate, and understand program paths, a common component of many static analysis tools' error reports. We performed a controlled user study to measure the benefit of Path Projection in triaging error reports from Locksmith, a data race detection tool for C. We found that Path Projection improved participants' time to complete this task, without affecting accuracy, and that participants felt Path Projection was useful.

1. Introduction

The research and industrial communities have made great strides in recent years developing sophisticated *defect detection* tools based on static analysis. Such tools have been used to find tens of thousands of bugs in real-world systems, including Linux [17, 16, 14], Microsoft productivity software [25, 30], and Java software [31], and in some cases have been used to verify the correctness of mission critical systems [6, 13]. Large companies such as Microsoft, Google, eBay, and Goldman-Sachs are now integrating tools into their regular development processes [10, 27, 37], and companies like Coverity [8] and Fortify [19], which provide defect detection tools and services, have hundreds of customers.

Most research on defect detection tools has focused on designing new static analysis *algorithms*. We believe it is equally important to study the other aspects of static analysis *tools*. Indeed, Pincus states that *"Actual analysis is only a small part of any program analysis tool [used at Microsoft]. In PREfix, [it is] less than 10% of the 'code mass'."* [33].

Generally speaking, static analysis tool users must perform two tasks: *triage*, deciding whether a report is a true or false positive, and *remediation*, fixing a true bug. An effective tool will assist the engineer in performing these tasks. However, while many tools provide support for categorizing and managing error reports, most provide little assistance for determining whether a report is true or false, and if true, how to fix it.

To address this problem, we present a new user interface toolkit called *Path Projection* that helps users visualize, navigate, and understand *program paths*—call stacks, control flow paths, or data flow paths—which are a common component of many static analysis tools' error reports. Our toolkit accepts an XML error report containing a set of paths, each consisting of a sequence of program statements, and automatically produces a concise view of them in conjunction with the source code. Path Projection aims to help engineers understand error reports more easily, to improve the speed and accuracy of triage and remediation.

An underlying principle of Path Projection's design is to keep the path display as close to the original source code layout as possible, since that is what the programmer is most familiar with. To follow this principle, we use three main techniques. We perform *function call inlining* to textually insert the bodies of called functions just below the call site, which rearranges the source code in the order of the path. We perform *code folding* to automatically hide potentially-irrelevant statements that are not involved in the path. Finally, we show multiple paths *side-by-side* for easy comparison. While other interfaces incorporate some of these features, we believe Path Projection's combination and design of features is novel.

We evaluated Path Projection's utility by performing a controlled experiment in which users triaged reports produced by Locksmith, a static data race detection tool for C. When Locksmith finds a potential data race, it produces an error report that includes a set of call stacks (the paths). Each call stack describes a concurrently-executing thread that contains an access involved in the potential race. For Locksmith, the triaging task requires examining each reported call stack to decide whether it is actually *realizable* at run time, and then deciding whether there are at least two realizable accesses that can execute in parallel.

To our knowledge, ours is the first work to empirically study a user interface for defect detection tools using sophisticated static analysis. While commercial vendors may study the utility of their interfaces internally, no results of such studies are publicly available. Independent evaluation of commercial tools is also difficult because of the tools' licensing, which often forbids disclosure of information.

In our study, we measured users' completion time and accuracy in triaging Locksmith reports, comparing Path Projection to a "standard" viewer that we designed to include the textual error report along with commonly-used IDE features. Both interfaces included a *checklist* specialized to each error report that enumerates the sub-tasks needed to triage the report correctly. We did not require that users propose actual fixes to the code, since even when a bug is clear its proper fix may be hard to determine.

In our within-subjects study, each user participated in one session with one interface, and one session with the other interface. Half the participants started with Path Projection, and the other half began with the standard viewer, to help factor out learning effects. In each session, after some introductory material, the user was asked to triage three error reports. At the end of the experiment, we asked users to qualitatively evaluate the interface and compare both.

We found that Path Projection improved the time it takes to triage a bug, and participants using it made about the same number of mistakes as with the standard viewer. Moreover, in Path Projection users spent little time looking at the error report itself. This suggests that Path Projection succeeds in making paths easy to see

[0] Technical Report CS-TR-4919, Department of Computer Science, University of Maryland, College Park, August 2008.

and understand in the source code view. Users clearly preferred Path Projection over the standard viewer, and generally rated all the features of Path Projection as somewhat or very useful. We also observed that the checklist dramatically reduced the overall triaging times for users of both interfaces, compared to an earlier pilot study. Though this result is not scientifically rigorous (several other features changed between the pilot and the current study), we believe it suggests checklists would be a useful addition to many static analysis interfaces, and merit further study.

In summary, this paper makes two main contributions:

1. We present *Path Projection*, a novel toolkit for visualizing program paths (Section 3). While mature static analysis tools can have sophisticated graphical user interfaces (Section 7), these interfaces are designed for particular tools and cannot easily be used in other contexts. We show how to apply Path Projection both to Locksmith and to BLAST [5], a software model checking tool. We believe Path Projection's combination and design of user interface features is novel.
2. We present quantitative and qualitative evidence of Path Projection's benefits in triaging Locksmith error reports (Sections 5 and 6). To our knowledge, ours is the first study to consider the task of triaging defect detection tool error reports, and the first to consider the user interface in this context. Our study results provide some scientific understanding of which features are most important for making users more effective when using static analysis tools.

We believe Path Projection is a valuable new toolkit that can benefit a wide range of static analyses.

2. Background: Locksmith

Locksmith [34], our target static analysis, works by enforcing the *guarded-by pattern* [38]: for each memory location shared among threads, there must exist a lock that is consistently acquired at every access to that location.[1] Locksmith initially performs an *alias analysis* to model the pointers and locks in the program. It then uses a *sharing analysis* to determine what locations are thread-shared, and a *lock state analysis* to compute the set of locks that guard each program point. Finally, Locksmith issues an error report for any shared location that is inconsistently guarded.

A sample Locksmith error report is shown in the pane labeled (1) in Figure 1 (we will discuss the rest of this figure later). This error report comes from `aget`, a multithreaded FTP client we examined previously [34]. The report lists the shared variable involved in the race (`prev`); a set of call stacks leading to conflicting accesses to the shared variable (indicated by "dereference" at the end of each stack); and the locks held during the accesses. In this example, the first call stack reaches the offending access via a thread created in `main`, whereas the second call stack reaches the same offending access via a different thread created in `resume_get`. In both cases, there are no locks held at the access.[2]

A key feature of Locksmith is that it aims to be *sound*, meaning if it reports no data races in the input program, then that program is indeed race free. However, this necessarily leads to some conservatism in the analysis and may result in *false positives*, which are reports that do not correspond to actual races in the code. In particular, Locksmith assumes that any branch in the code could potentially be taken. Thus, as part of triaging this error report, the user must check that the potentially-racing accesses are *simultaneously realizable*, meaning there is some program execution in which both accesses could happen at the same time.

To find such an execution, the user must examine the control flow logic along each reported path and ensure it does not preclude both paths from occurring at the same time. Tracing through several control flow paths at once can be quite challenging to do by hand, especially if the process involves examining more than a screenful of source code. This kind of path tracing task is actually common across many different static analysis tools (Section 7). The focus of Path Projection and our empirical study is on improving user performance on this task in particular, and when we refer to *triaging* in the remainder of the paper, we will mean this task applied to Locksmith.

Besides unrealizable paths, there are several other reasons Locksmith may report a false positive, e.g., the sharing analysis may incorrectly deem two unrelated variables as shared, or the lock state analysis may miss locks that are actually held. For many programs that we have looked at, Locksmith's lock state analysis is accurate, and warnings often involve only simple sharing (e.g., of global variables), so our current work focuses on tracing program paths. In general, these other cases are certainly important, and we intend to address them in future work, for example, using techniques from MrSpidey [18] or CQual [20].

Example scenario Figure 1 shows a screenshot of our *standard viewer*, which displays the Locksmith error report (1) side-by-side with syntax-colored program source code (2). This interface is our baseline for comparing against Path Projection, and represents the assistance a typical editor or IDE gives users in understanding Locksmith error reports.

The first step in triaging is to identify pairs of accesses that are not guarded by the same locks (or are guarded by no locks), and thus could result in a race. This is trivial given the error report, and in fact we automatically identify these pairs and generate a checklist (3). We will focus on tab "1–2" to check for potential races between threads in paths 1 and 2. We defer further discussion of the checklist to Section 4.

We need to trace through the control-flow of the program to see under what conditions the two dereferences are reachable. The call stacks in the error report show exactly what code to examine. In the screenshot in Figure 1, we are tracing path 1, and we have split the source code view horizontally to show two regions of code at once (4). We begin by tracing path 1 from the thread creation site, which can be reached by clicking on the hyperlink (5), and look for any branch conditions that may prevent the thread from being created. We can continue tracing to the access by clicking on the hyperlinks or scrolling through the code. As we navigate from function to function, we can further split windows to keep various context together on the screen, and close splits when the window becomes too crowded.

After tracing path 1, we need to trace through path 2 in the same way, and ultimately decide whether both paths are simultaneously realizable. This is quite tedious, even though `aget` is slightly less than 2,000 lines of code. Worse, this is only one of 62 warnings that Locksmith reports for `aget`, and the procedure has to be repeated for each warning.

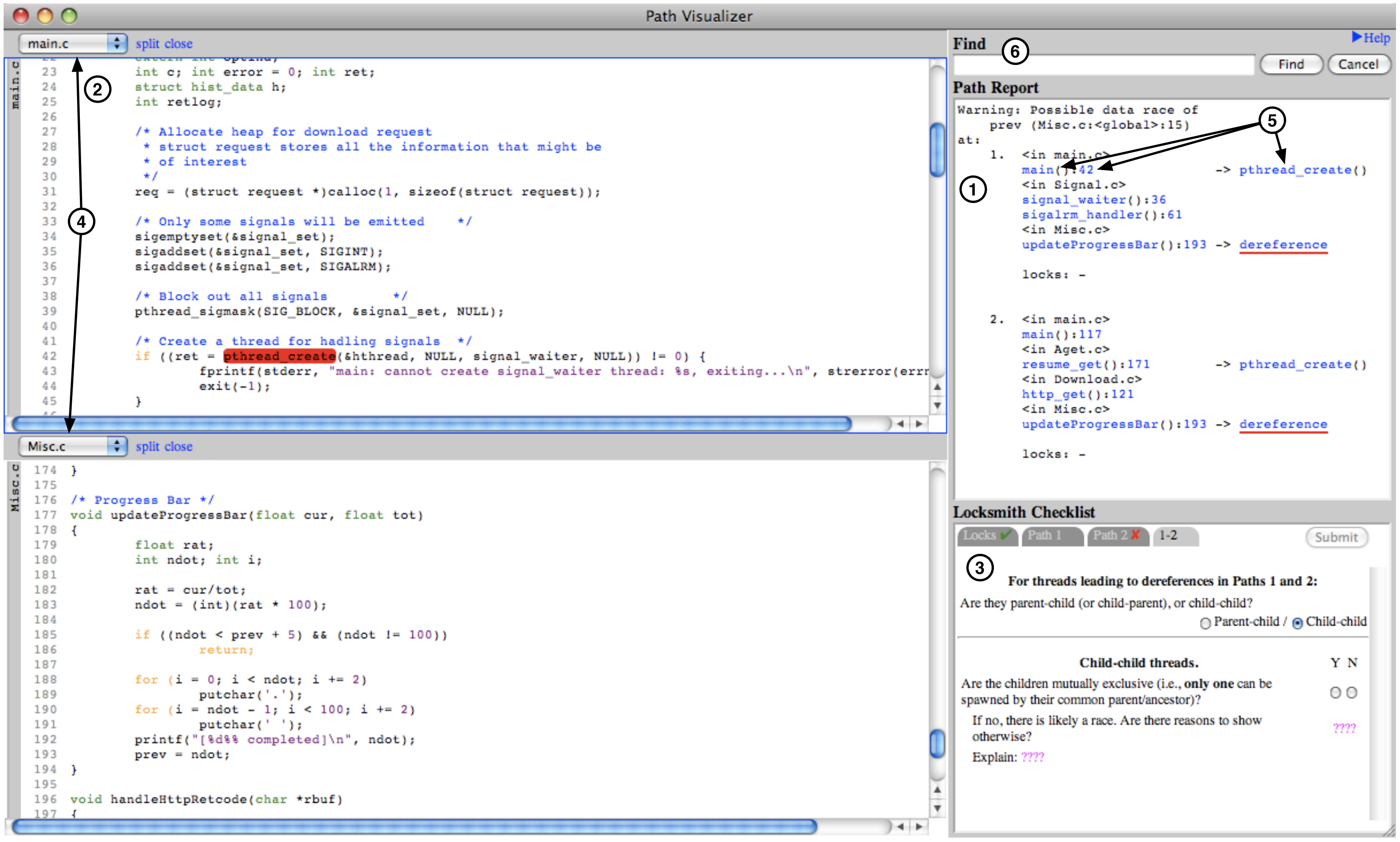
3. Path Projection

Program paths are commonly used by static analysis tools to report errors. However, as the previous section illustrates, tracing program paths becomes difficult when the user needs to examine discontinuous lines of code across many different files.

Standard code editors provide some support for tracing program paths. For example, they typically allow the user to view more

[1] Locksmith also includes logic to support thread-local and read-only variables (which need not be protected by locks), as well as initialization (prior to becoming shared, variables may be written to without holding a lock).

[2] Note that this report format, used in our experiments (Section 6), is slightly different than Locksmith's current output, but the differences are merely syntactic.



```
23      int c; int error = 0; int ret;
24      struct hist_data h;
25      int retlog;
26
27      /* Allocate heap for download request
28       * struct request stores all the information that might be
29       * of interest
30       */
31      req = (struct request *)calloc(1, sizeof(struct request));
32
33      /* Only some signals will be emitted     */
34      sigemptyset(&signal_set);
35      sigaddset(&signal_set, SIGINT);
36      sigaddset(&signal_set, SIGALRM);
37
38      /* Block out all signals         */
39      pthread_sigmask(SIG_BLOCK, &signal_set, NULL);
40
41      /* Create a thread for hadling signals  */
42      if ((ret = pthread_create(&hthread, NULL, signal_waiter, NULL)) != 0) {
43              fprintf(stderr, "main: cannot create signal_waiter thread: %s, exiting...\n", strerror(err
44              exit(-1);
45      }
```

```
174 }
175
176 /* Progress Bar */
177 void updateProgressBar(float cur, float tot)
178 {
179         float rat;
180         int ndot; int i;
181
182         rat = cur/tot;
183         ndot = (int)(rat * 100);
184
185         if ((ndot < prev + 5) && (ndot != 100))
186                 return;
187
188         for (i = 0; i < ndot; i += 2)
189                 putchar('.');
190         for (i = ndot - 1; i < 100; i += 2)
191                 putchar(' ');
192         printf("[%d%% completed]\n", ndot);
193         prev = ndot;
194 }
195
196 void handleHttpRetcode(char *rbuf)
197 {
```

```
Warning: Possible data race of
    prev (Misc.c:<global>:15)
at:
    1.  <in main.c>
        main():42                   -> pthread_create()
        <in Signal.c>
        signal_waiter():36
        sigalrm_handler():61
        <in Misc.c>
        updateProgressBar():193 -> dereference

        locks: -


    2.  <in main.c>
        main():117
        <in Aget.c>
        resume_get():171            -> pthread_create()
        <in Download.c>
        http_get():121
        <in Misc.c>
        updateProgressBar():193 -> dereference

        locks: -
```

For threads leading to dereferences in Paths 1 and 2:

Are they parent-child (or child-parent), or child-child?

Parent-child / [✓ Child-child]

Child-child threads. Y N

Are the children mutually exclusive (i.e., **only one** can be spawned by their common parent/ancestor)? ○ ○

If no, there is likely a race. Are there reasons to show otherwise? ????

Explain: ????

Figure 1. Standard viewer (picture in color; labels described in text)

than one file to be by opening multiple windows or by splitting a window. Code folding is another commonly available feature used to hide irrelevant code.

The key issue, however, is that the user has to invoke and manage these features manually. In particular, the user has to carefully consider the tradeoffs of these features, e.g., after opening or splitting windows, the user may have to move, resize or close other windows to make them visible or simply to reduce screen clutter.

This places a significant cognitive burden on the user to extract and organize relevant information from the source code and the error report, and distracts from the actual task of understanding a program path. If paths are long or complicated, as they often are, it can be hard to keep track of the context of the path while managing windows and folded code. In our experience, it is all too easy to get lost on a long path and have to backtrack or retrace it many times.

3.1 Design Guidelines

To develop a better interface for tracing program paths, we can look to guidelines developed by researchers in information visualization. We found three strategies described by Card et al. [7] to be particularly applicable for our task:

1. *Increase users' memory and processing resources and reduce the search for information.* The Locksmith error report is compact, but examining the path in the actual source code may involve many different source code lines spread across many different files. We would ideally like to put all the necessary information for triaging an error report on one screen, so the user can see all the information at the same time. We would also like to allow the user to hide any unimportant information, to further reduce the cognitive burden.
2. *Use visual representation to enhance pattern detection.* We would like to visually distinguish the source code lines appearing in the Locksmith error report from the other lines in the program, since the lines in the error reports are presumably very important. We would also like to bring important threading API calls, e.g., invocations of pthread_X functions, to the user's attention.
3. *Encode information in a manipulable medium.* We need to give the user good mechanisms for searching and comparing the information we present to them.

Another key guideline we would like to follow is that "code should look like code." We want the user to be able to relate any visualization back to the original source code. This is based on our experience as programmers: we spend a large fraction of time editing source code in standard textual form, and relatively little working with abstract visualizations. Understanding Locksmith error reports requires looking at source code in great detail. A visualization that looks like source code will be familiar, which should increase acceptance and could increase comprehension.

3.2 Interface Features

Figure 2 shows the Path Projection interface for the same program shown in the standard viewer in Figure 1. The core feature of Path Projection is that it makes multiple program paths manifest in the display of the source code. To achieve this, Path Projection uses three main techniques:

Function call inlining Path Projection inlines function calls along a path. In the left path in the example, we see that the bodies of signal_waiter (called by pthread_create), sigalrm_handler, and updateProgressBar are displayed in a series of nested boxes immediately below the calling line (1). We color and indent each box to help visually group the lines from each function (Principle 2). This feature is where our interface name comes from—we are "projecting" the source code onto the error path. Our motivation is to reduce the need for the programmer to look at the error report, since the path is apparent in the visualization. We also underline the racing access (innermost boxes) to make it easy to identify.

Notice that in Path Projection, we actually visually rearrange function bodies in the original source code to conform to the order they appear in the path. Moreover, we may pull in functions from multiple files (in our example, three separate files) and display them together. Together, these regroupings help keep relevant information close together to reduce search and free up cognitive resources (Principle 1).

In the standard viewer, the user could achieve a similar result by splitting the display into several pieces to show the functions along the path. However, this quickly becomes tedious if there are several functions to show at once, and it adds significant mechanical overhead to viewing and navigating a path. Moreover, without code folding, which we discuss next, it can be difficult to see all the relevant parts of the functions simultaneously.

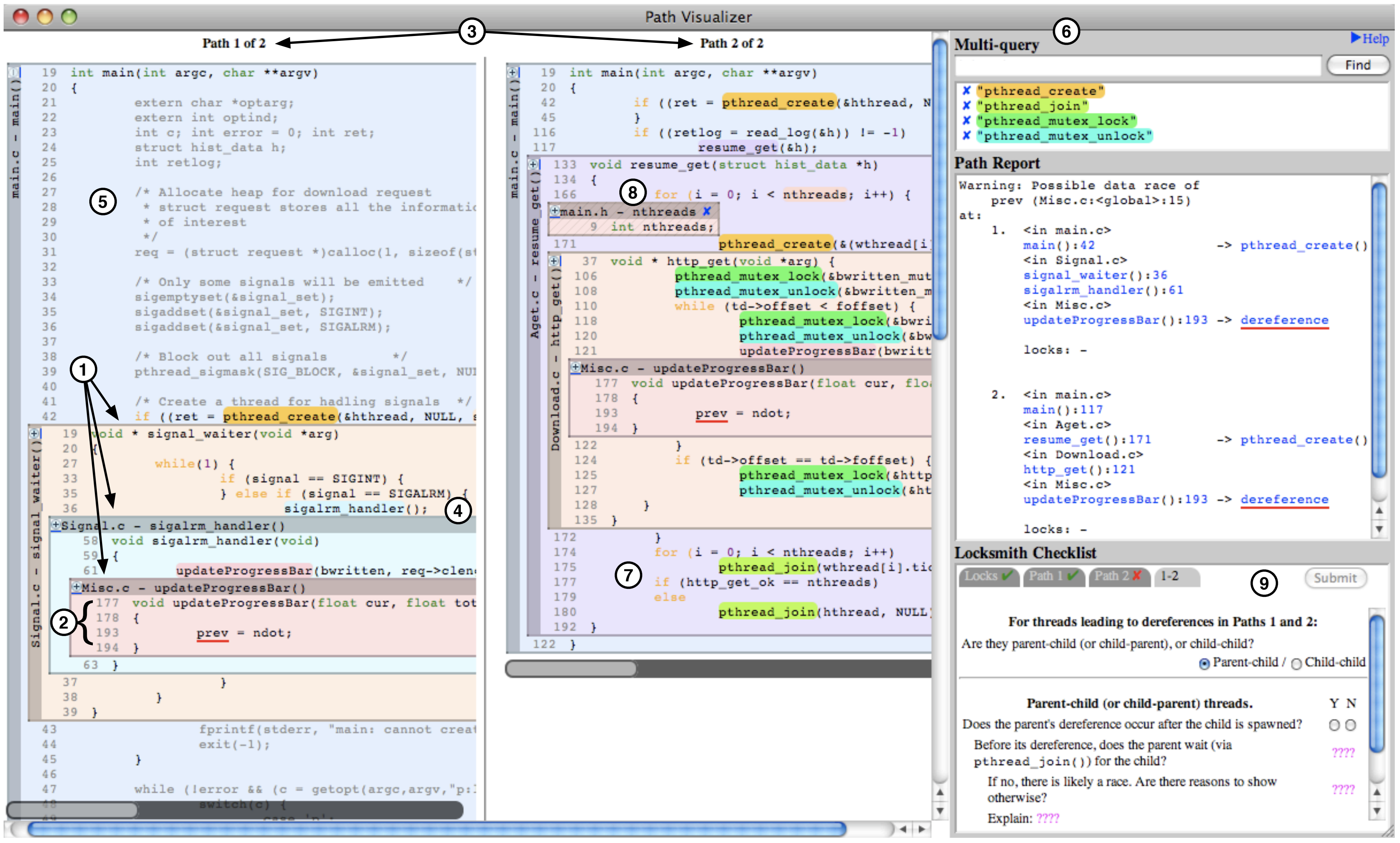
Code folding To keep as much relevant information on one screen as possible (Principle 1), we filter by default irrelevant statements from the displayed source code. For example, in the function updateProgressBar in the left path, we have folded away all lines (2) except the implicated access (line 193) and the enclosing lexical elements (the function definition and open and close curly braces). The user can tell code has been folded by noticing non-consecutive line numbering.

Our code folding algorithm is a syntax-based heuristic that tags lines in the program as either *relevant* or *irrelevant*. Irrelevant lines are hidden when the code is folded. For each line l listed in the error report, we tag as relevant line l itself; the opening and closing statements of any lexical blocks that enclose l; and the beginning and end of the containing function. For lexical blocks that include a guard (i.e., if, for, while, or do), we include the guard itself, and for an if block, we show the line containing the else (though not the contents of the else block). For example, the call to sigalrm_handler (4) is in the error path, and so we reveal it, the closing if and while blocks, and the beginning and end of the signal_waiter function. Our code folding heuristic resembles the degree-of-interest model proposed by Furnas [21]. A more sound approach would be to use program slicing [41] to determine relevant lines; and in a sense, our code folding algorithm can be thought of as an extremely simple but imprecise slicing technique.

Since our code folding algorithm is heuristic and may hide elements that the user actually needs to examine, we allow the user to click the expansion button in the upper-left corner of a box to reveal all code in the corresponding function. When unfolded, source lines tagged as irrelevant are colored gray to ensure the path continues to stand out. For example, we have unfolded the body of main (5) in Figure 2.

While code folding is common in many IDEs, most require the user manually perform folding on individual lexical blocks, which can be time consuming and tricky to get exactly right. In contrast, we use the path to automatically decide which lines of code to reveal or fold away, requiring no user effort. Furthermore, since we display each path in a separate column, we can apply our code folding algorithm to each path individually.

Side-by-side paths This error report contains two paths, each of which is displayed side-by-side in its own column ((3), left side of Figure 2). This parallel display makes relationships between the paths easier to see than in the standard interface, which would require flipping between different views (Principle 1). Each column is capped at a maximum width, and users can horizontally scroll columns individually. This helps prevent files that are unusually wide from cluttering the display.



```
19  int main(int argc, char **argv)
20  {
21          extern char *optarg;
22          extern int optind;
23          int c; int error = 0; int ret;
24          struct hist_data h;
25          int retlog;
26
27          /* Allocate heap for download request
28           * struct request stores all the informatic
29           * of interest
30           */
31          req = (struct request *)calloc(1, sizeof(st
32
33          /* Only some signals will be emitted     */
34          sigemptyset(&signal_set);
35          sigaddset(&signal_set, SIGINT);
36          sigaddset(&signal_set, SIGALRM);
37
38          /* Block out all signals          */
39          pthread_sigmask(SIG_BLOCK, &signal_set, NU
40
41          /* Create a thread for hadling signals  */
42          if ((ret = pthread_create(&hthread, NULL, s
    19  void * signal_waiter(void *arg)
    20  {
    27          while(1) {
    33                  if (signal == SIGINT) {
    35                  } else if (signal == SIGALRM) {
    36                      sigalrm_handler();
        58  void sigalrm_handler(void)
        59  {
        61          updateProgressBar(bwritten, req->clen
            177  void updateProgressBar(float cur, float tot
            178  {
            193          prev = ndot;
            194  }
        63  }
    37                  }
    38          }
    39  }
43                  fprintf(stderr, "main: cannot creat
44                  exit(-1);
45          }
46
47          while (!error && (c = getopt(argc,argv,"p:
48                  switch(c) {
```

```
19  int main(int argc, char **argv)
20  {
42          if ((ret = pthread_create(&hthread, N
45          }
116         if ((retlog = read_log(&h)) != -1)
117                 resume_get(&h);
    133  void resume_get(struct hist_data *h)
    134  {
    166      for (i = 0; i < nthreads; i++) {
             9  int nthreads;
    171              pthread_create(&(wthread[i]
         37  void * http_get(void *arg) {
         106         pthread_mutex_lock(&bwritten_mut
         108         pthread_mutex_unlock(&bwritten_m
         110         while (td->offset < foffset) {
         118                 pthread_mutex_lock(&bwri
         120                 pthread_mutex_unlock(&bw
         121                 updateProgressBar(bwritt
             177  void updateProgressBar(float cur, flo
             178  {
             193          prev = ndot;
             194  }
         122         }
         124         if (td->offset == td->foffset) {
         125                 pthread_mutex_lock(&http
         127                 pthread_mutex_unlock(&ht
         128         }
         135 }
    172      }
    174      for (i = 0; i < nthreads; i++)
    175              pthread_join(wthread[i].tic
    177      if (http_get_ok == nthreads)
    179      else
    180              pthread_join(hthread, NULL)
    192  }
122  }
```

Multi-query

- ✗ "pthread_create"
- ✗ "pthread_join"
- ✗ "pthread_mutex_lock"
- ✗ "pthread_mutex_unlock"

Path Report

```
Warning: Possible data race of
      prev (Misc.c:<global>:15)
at:
    1.   <in main.c>
         main():42                -> pthread_create()
         <in Signal.c>
         signal_waiter():36
         sigalrm_handler():61
         <in Misc.c>
         updateProgressBar():193 -> dereference

         locks: -


    2.   <in main.c>
         main():117
         <in Aget.c>
         resume_get():171         -> pthread_create()
         <in Download.c>
         http_get():121
         <in Misc.c>
         updateProgressBar():193 -> dereference

         locks: -
```

Locksmith Checklist

Locks ✓ Path 1 ✓ Path 2 ✗ 1-2

For threads leading to dereferences in Paths 1 and 2:

Are they parent-child (or child-parent), or child-child?

[✓ Parent-child] / Child-child

Parent-child (or child-parent) threads. Y N

Does the parent's dereference occur after the child is spawned? ○ ○

Before its dereference, does the parent wait (via pthread_join()) for the child? ????

If no, there is likely a race. Are there reasons to show otherwise? ????

Explain: ????

Figure 2. Path Projection (picture in color; labels described in text). This and additional screenshots can be found at at http://www.cs.umd.edu/projects/PL/PP.

```
1  Trace:
2    In function a (
3      On line a:5, call b (
4        On line b:10, call c (
5          On line c:15, call d
6        )
7      )
8      On line a:20, call e (
9        On line e:25, call f
10     )
11   )
```

```
1  Trace:
2    In function <path name="a">a (
3      On line <detour line="5" name="b">a:5, call <path name="b">b (
4        On line <detour line="10" name="c">b:10, call <path name="c">c (
5          On line c:15, call d
6        )</path></detour>
7      )</path></detour>
8      On line <detour line="20" name="e">a:20, call <path name="e">e (
9        On line e:25, call f
10     )</path></detour>
11   )</path>
```

Figure 3. A textual path-based report (left) can be converted into the pathreport XML format (right) using `<path>` and `<detour>` (some attributes omitted for brevity).

Triaging Locksmith's error report requires comparing more than one path, making this feature a necessity. However, we believe that side-by-side paths would be useful for other tools too. For example, a model-checker may display known-good paths for comparison with the error path.

In our experiments, participants used a wide-screen monitor to make it easier to see multiple paths simultaneously. Since wide-screen displays are commonly available and popular, we think designing an interface with such displays in mind is reasonable.

Additional interface features Path Projection includes several other features to make it easier to use. We include a *multi-query* search facility (6) that allow users to locate multiple terms and keep them highlighted at the same time. Each term is distinguished by a different color in the source code, and the user can cancel the highlighting of any term individually. Any source line containing a match is automatically marked as relevant, as are any lexical blocks enclosing the match. For example, in the screenshot in Figure 2, calls to pthread_join, which are not included in the error report, have been marked as relevant due to the search (7). This facility follows Principle 3, since it allows the user to manipulate code folding in a natural way. In our experiments, we initialized multi-query to find the four pthread functions shown in the screenshot, since in our pilot study we found users almost always want to locate uses of these functions.

Our interface also includes a *reveal definition* facility that uses inlining to show the definition of a function or variable. In the screenshot, the user has clicked on nthreads, and its definition has been inlined below the use (8). While this feature seems potentially handy, we found it was rarely used by participants in our experiments.

Lastly, Path Projection still includes the original error report from which the visualization was generated, to act as a back-up and to provide consistency with the standard view. As with the standard view, the report is hyperlinked to the source display.

3.3 Applying Path Projection to Other Tools

We intend Path Projection to be general toolkit that may be used by static analysis tool developers to visualize their error reports. To this end, Path Projection is implemented as a standalone tool that takes as input an XML-based error report and the source code under analysis.

XML format Our XML format, called pathreport, describes one or more program path using three kinds of tags. It is designed to easily convert a textual path-based report by marking it up appropriately. Briefly, the three tags are:

`<path>` marks text that corresponds to a continuous block of code, such as a function. A `<path>` can contain any number of `<detour>` and `<marker>` tags.

`<detour>` marks text that corresponds to a reference in some code, such as a call site. The parent `<path>` provides the context for this reference, e.g., the function in which the call site appears. Each `<detour>` tag must in turn contain exactly one `<path>` tag.

`<marker>` tags mark any other texts, such as function names, variable names, or line numbers, that point to code that may be interesting.

These tags also require attributes to precisely indicate information such as the file where a function is defined. Together, `<path>` and `<detour>` recursively describe a program path, and `<marker>` can be used to mark any interesting line of code along the path.

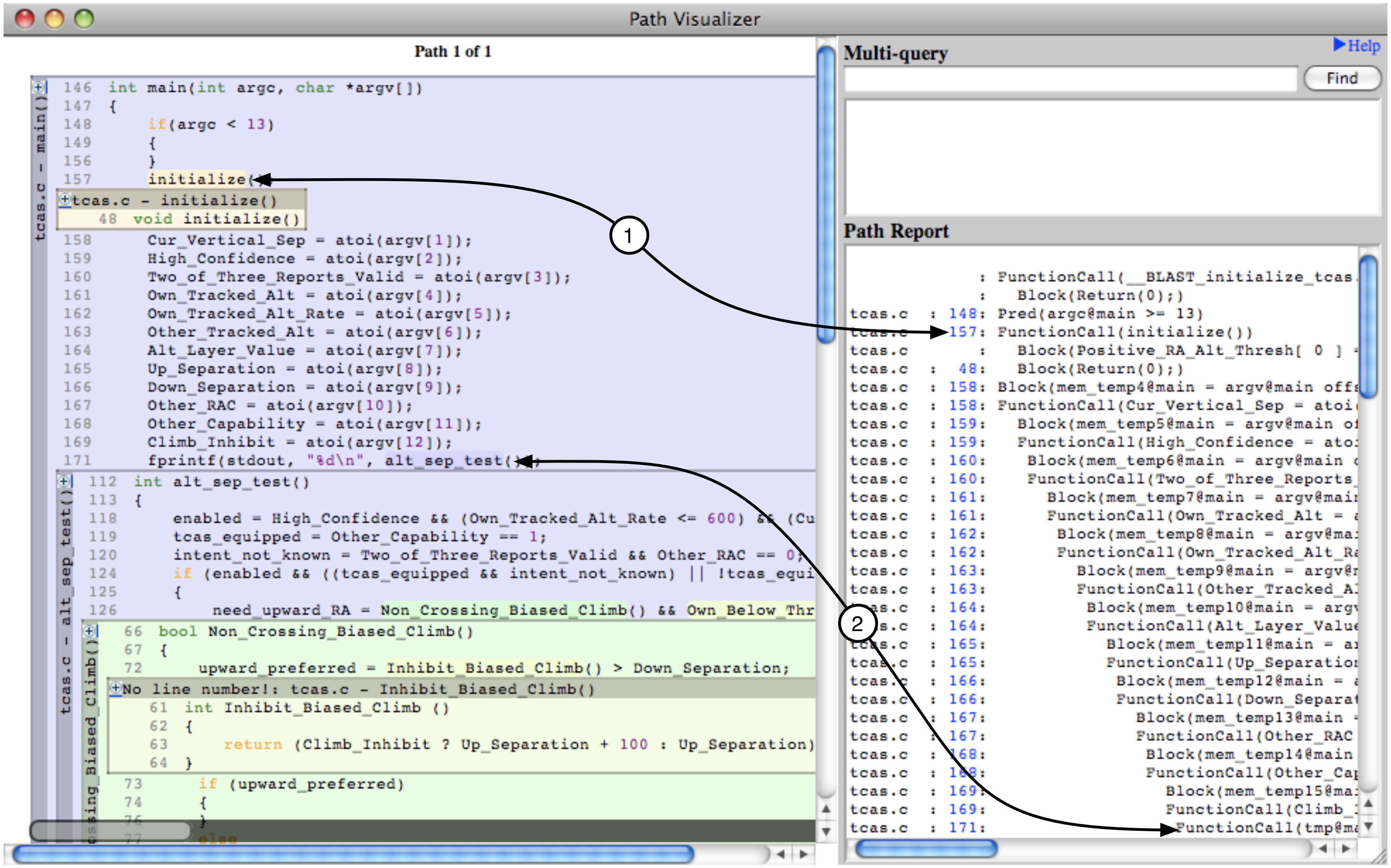
Figure 3 shows how a path-based textual report can be marked up with `<path>` and `<detour>`. Note that on line 2, `<path>` marks the entire call to a. Within that, two `<detour>` tags on line 3 and line 8 mark the call sites to b and e that appear in a.

Path Projection on BLAST In addition to Locksmith, we have applied Path Projection to counterexample traces produced by BLAST [5], a software model checking tool. Such traces are not call stacks, as in Locksmith, but rather are execution traces that include function calls and returns. We use a short awk script to post-process a BLAST trace into our XML format. We reformat the line annotations from the original report for brevity, but retain the indentations as they reveal relationships between implicated lines.

Figure 4 shows an example of BLAST in Path Projection, generated from a post-processed counterexample trace that describes how a failing assertion (not shown) can be reached. The counterexample trace shown is 102 lines long, much longer than is typical with Locksmith. Note that BLAST's counterexample trace is not a call stack, as in Locksmith, but an execution trace that includes function calls and returns.

We find the error report confusing in its textual form, e.g., line 4 implicates a call to initialize in line 157 of tcas.c, and lines 5-6 are indented to indicate the function body; but the indented lines after line 8 do not correspond to the body of atoi. As such, it can be quite difficult to match function calls and returns in a long error report. Occasionally, there are also missing line numbers, such as on line 5 in the error report. Furthermore, it is not obvious from the error report that many implicated lines are actually not interesting.

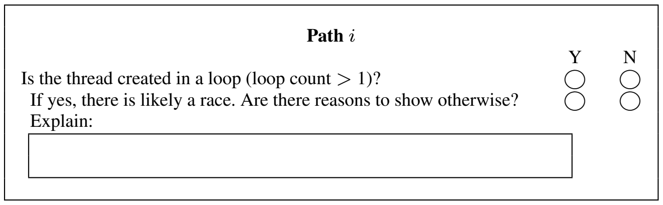
Path Projection makes the structure of the path much clearer. We can quite easily tell that the beginning of the path contains a call to initialize (1), and that the subsequent 24 lines in the error report correspond to the input program's command-line interface. The next



```
tcas.c - main()
146 int main(int argc, char *argv[])
147 {
148     if(argc < 13)
149     {
156     }
157     initialize();
tcas.c - initialize()
 48 void initialize()
158     Cur_Vertical_Sep = atoi(argv[1]);
159     High_Confidence = atoi(argv[2]);
160     Two_of_Three_Reports_Valid = atoi(argv[3]);
161     Own_Tracked_Alt = atoi(argv[4]);
162     Own_Tracked_Alt_Rate = atoi(argv[5]);
163     Other_Tracked_Alt = atoi(argv[6]);
164     Alt_Layer_Value = atoi(argv[7]);
165     Up_Separation = atoi(argv[8]);
166     Down_Separation = atoi(argv[9]);
167     Other_RAC = atoi(argv[10]);
168     Other_Capability = atoi(argv[11]);
169     Climb_Inhibit = atoi(argv[12]);
171     fprintf(stdout, "%d\n", alt_sep_test());
tcas.c - alt_sep_test()
112 int alt_sep_test()
113 {
118     enabled = High_Confidence && (Own_Tracked_Alt_Rate <= 600) && (Cu
119     tcas_equipped = Other_Capability == 1;
120     intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == 0;
124     if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equi
125     {
126         need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Thr
ssing_Biased_Climb()
 66 bool Non_Crossing_Biased_Climb()
 67 {
 72     upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
No line number!: tcas.c - Inhibit_Biased_Climb()
 61 int Inhibit_Biased_Climb ()
 62 {
 63     return (Climb_Inhibit ? Up_Separation + 100 : Up_Separation)
 64 }
 73     if (upward_preferred)
 74     {
 76     }
 77     else
```

```
               : FunctionCall(__BLAST_initialize_tcas.
               :   Block(Return(0);)
tcas.c  : 148: Pred(argc@main >= 13)
tcas.c  : 157: FunctionCall(initialize())
tcas.c  :    :   Block(Positive_RA_Alt_Thresh[ 0 ] =
tcas.c  :  48:   Block(Return(0);)
tcas.c  : 158: Block(mem_temp4@main = argv@main off
tcas.c  : 158: FunctionCall(Cur_Vertical_Sep = atoi
tcas.c  : 159:   Block(mem_temp5@main = argv@main of
tcas.c  : 159:   FunctionCall(High_Confidence = ato
tcas.c  : 160:    Block(mem_temp6@main = argv@main
tcas.c  : 160:    FunctionCall(Two_of_Three_Reports_
tcas.c  : 161:     Block(mem_temp7@main = argv@mai
tcas.c  : 161:     FunctionCall(Own_Tracked_Alt = a
tcas.c  : 162:      Block(mem_temp8@main = argv@ma
tcas.c  : 162:      FunctionCall(Own_Tracked_Alt_Ra
tcas.c  : 163:       Block(mem_temp9@main = argv@r
tcas.c  : 163:       FunctionCall(Other_Tracked_Al
tcas.c  : 164:        Block(mem_temp10@main = argv
tcas.c  : 164:        FunctionCall(Alt_Layer_Value
tcas.c  : 165:         Block(mem_temp11@main = ar
tcas.c  : 165:         FunctionCall(Up_Separation
tcas.c  : 166:          Block(mem_temp12@main = a
tcas.c  : 166:          FunctionCall(Down_Separat
tcas.c  : 167:           Block(mem_temp13@main =
tcas.c  : 167:           FunctionCall(Other_RAC
tcas.c  : 168:            Block(mem_temp14@main
tcas.c  : 168:            FunctionCall(Other_Cap
tcas.c  : 169:             Block(mem_temp15@ma
tcas.c  : 169:             FunctionCall(Climb_
tcas.c  : 171:              FunctionCall(tmp@ma
```

Figure 4. Path Projection applied to a post-processed counterexample trace produced by BLAST (picture in color; labels described in text).



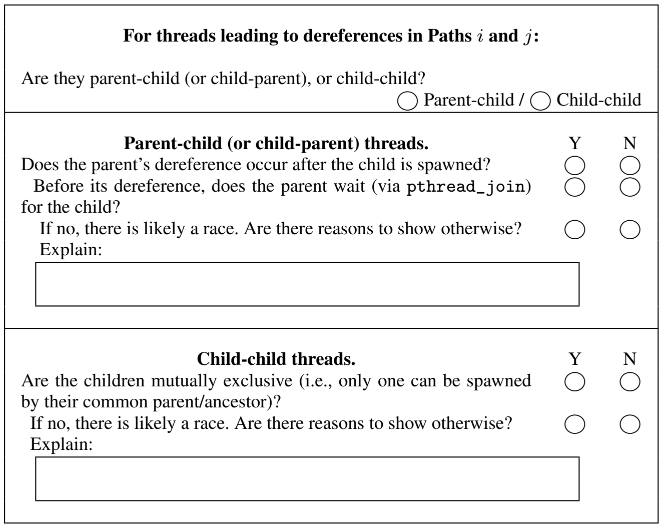
Path i

Is the thread created in a loop (loop count > 1)? Y ◯ N ◯

If yes, there is likely a race. Are there reasons to show otherwise? Y ◯ N ◯

Explain:

(a) Single path i



For threads leading to dereferences in Paths i and j:

Are they parent-child (or child-parent), or child-child?

◯ Parent-child / ◯ Child-child

Parent-child (or child-parent) threads.

Does the parent's dereference occur after the child is spawned? Y ◯ N ◯

Before its dereference, does the parent wait (via `pthread_join`) for the child? Y ◯ N ◯

If no, there is likely a race. Are there reasons to show otherwise? Y ◯ N ◯

Explain:

Child-child threads.

Are the children mutually exclusive (i.e., only one can be spawned by their common parent/ancestor)? Y ◯ N ◯

If no, there is likely a race. Are there reasons to show otherwise? Y ◯ N ◯

Explain:

(b) Pair of paths i and j

Figure 5. Checklist tabs

interesting line is a call to alt_sep_text (2), which is implicated further down the report, on the last line visible in Figure 4 (obscured due to indentation). However, this is obvious at first glance in Path Projection.

We believe that this example shows how Path Projection can be quite useful to understand complicated error reports. We think that using Path Projection can also suggest ways to improve error reports, e.g., BLAST should not indent function calls with no corresponding body.

4. Checklist

In our pilot study we found that even with extensive pre-experiment tutorials, participants had trouble determining how to accomplish this triaging task, and would often get distracted with irrelevant features, such as locating the definition of a global variable. This inconsistent behavior confounded our attempts to measure the benefits of the PP interface, since it varied significantly depending on the participant. To address this issue, we developed a tabbed *checklist* that breaks down this task into smaller sub-tasks, one per tab, that identify conditions under which reported thread states, if individually realizable, could execute in parallel and thus constitute a true race. A single error report could identify several races, so users must complete all tabs, at which point they may click *Submit* to complete the triage process.

A checklist is automatically generated for each Locksmith error report, and is identical in both interfaces—(3) in Figure 1 and (9) in Figure 2. The first tab, labeled *Locks* (not shown), asks the users to document where the locks held at the end of each path were acquired. This tab is merely an experimental device: since users tend to examine the code before moving to the checklist, we insert this tab first to measure this initial startup period. The remaining tabs have two flavors, both illustrated in Figure 5.

The tab shown in Figure 5(a) is generated once for each path i that ends in an access with no locks held. The user is asked to check whether the access could occur in a thread created in a loop. If it could have, then the same access may occur in two different threads, constituting a race. For example, consider Path 2 in Figure 2. The write to prev (underlined in red) occurs in a thread created on line 171 of Aget.c. Notice that that line appears in a for loop that actually creates multiple threads. Thus, what Locksmith reports as Path 2 is actually a summary of nthreads total paths, all of which may reach the same access. Since no lock is held at the access, and it is a write, we have found a data race. The last part of this tab asks the user to look for any logic that would prevent the data race from occurring. For example, perhaps nthreads is always 1, so only one thread is spawned, or perhaps the given state is not realizable due to inconsistent assumptions about the branches taken on the path. While we could attempt to break this condition down into further subtasks, we chose this more open-ended question to reduce visual clutter and complexity.

The tab in Figure 5(b) is generated once for each pair of paths i and j that end with inconsistent sets of locks held. First, the user selects whether the two threads are in a parent-child or other (child-child) relationship. In the first case, the user must check (a) that the access in the parent occurs after the child is created, and (b) that there is no parent-child synchronization with pthread_join. Case (a) is necessary for the parent and child to access a location in parallel. Case (b) is necessary because even if the parent accesses the location after creating its child, it might wait for the child to complete before performing the access, making parallel access impossible. If both (a) and (b) are true, then there is likely a race, and the user is asked to look for other program logic that would prevent a race.

In the second case, a child-child relationship, the user checks whether the two children are mutually exclusive. For example, they may be spawned in different branches of an if statement, only one of which can execute dynamically. Again, if they are not mutually exclusive, the user looks for other logic that would prevent a race.

Though we did not carefully consider the benefits of the checklist experimentally, we observed anecdotally that participants' completion times for our current study were reduced in both variance and magnitude (by an average of 3:46 minutes, or 41%), compared to the pilot study. This improvement strongly suggests that a tool-specific checklist has independent value, and that tool builders might consider designing checklists for use with their tools. An interesting direction for future work would be to explicitly study the benefit of checklists in performing triaging.

5. Experimental Evaluation

We evaluated Path Projection's utility in a controlled user study. In our experiment, participants are asked to perform a series of triaging tasks, using either Path Projection (PP), described in Section 3, or the standard viewer (SV), described in Section 2. To eliminate bias due to participants' prior experience with editors, we implemented our own standard viewer instead of using an existing editor.

For each task, the participant is presented with a Locksmith error report for a program selected from a test corpus of open source programs. The participant's goal is to determine whether the error report constitutes an actual data race or whether it is a false positive. In this task, PP provides an advantage over SV in triaging if it is faster, easier, and/or more accurate.

5.1 Participants

We recruited a total of eight participants (3 undergraduate, 5 graduate) for this experiment via e-mail and word-of-mouth advertising

	Session 1			Session 2		
1	PP/1.1	PP/1.2	PP/1.3	SV/2.1	SV/2.2	SV/2.3
2	SV/1.1	SV/1.2	SV/1.3	PP/2.1	PP/2.2	PP/2.3

Figure 6. User interface/problem number schedules

in the UMD Computer Science Department. We required the participants to have prior experience with C and with multithreaded programming (though not necessarily in C). All participants had taken at least one college-level class that involved multithreaded programming. On a scale of 1 (no experience) to 5 (very experienced), participants rated themselves between 3 and 4 when describing their ability to debug data races. Two participants had previous experience in using a race detection tool (Locksmith and Eraser [38]).

5.2 Design

Each participant was asked to perform the triaging task in two sessions, first with one interface and then with the other. Thus, our experiment is a within-subjects design consisting of two conditions, using PP and using SV. Since participants experienced both interfaces, we were able to measure comparative performance, and we could ask participants to qualitatively compare the two interfaces.

A within-subjects design potentially suffers from order effects, since participants may be biased towards the interface given first. To compensate, we perform counterbalancing on the interface order: participants are randomly placed into one of the two schedules shown in Figure 6.

Participants in the first schedule use PP in the first session and SV in the second, whereas participants in the second schedule use SV first and PP second. However, all participants receive the same set of problems, numbered 1.1–2.3, in the same order, which allows us to directly compare our observations of each problem without the need to account for order effects.

5.3 Procedure

At the beginning of each session, we ask participants to review a short tutorial and quiz on pthreads and data races, as well as a tutorial on Locksmith with emphasis on the items in the triaging checklist (below, Section 4). Then we introduce the user interface (PP or SV) with another tutorial. We make sure that participants are familiar with each interface by encouraging them to try every feature and to triage a simple data race problem using the interface.

Following the tutorial is a single practice trial and three actual trials, all of which follow the same procedure. In each trial, we first ask participants to triage a real Locksmith error report generated from Locksmith's test corpus (below, Section 5.4). We log participants' mouse movements during the trial and measure the total time to completion. Triaging ends when participants complete and submit the triaging checklist. Immediately after, we present participants with the same problem and ask them to explain out loud the steps they took to verify the warning. This allows us to compare their descriptions with our expectations.

We do not tell participants whether their answers are correct, for several reasons. Firstly, it is difficult for the experimenter to quickly judge the correctness of the answer, since identifying a data race may involve understanding many subtle aspects in a program. Secondly, we do not want the participants to become reliant on receiving an answer or to guess at an answer. Finally, in a real triaging scenario, users will not have the benefit of an oracle to confirm their intuitions.

We have found this two-stage procedure to be very effective in our pilot studies. In particular, it allows us to ask about specific interesting behaviors observed without interrupting the participant during the task. The participants also benefit from a limited form of feedback. By recalling their work to the experimenter, they can confirm their initial understanding, or notice mistakes made. Furthermore, we have found that participants gain a better understanding of the user interface by demonstrating it to another person. This also helps mitigate the effects of what turned out to be a long learning curve in triaging error reports. The experimenter may also ask for clarification to certain points, which may reveal inconsistencies or mistakes.

After the experiment, participants complete a questionnaire and are interviewed to determine their opinion of the user interface. Users are asked to evaluate each tool based on ease-of-learning, ease-of-use, and effectiveness in supporting the triaging task.

We ran the experiment on Mac OS X 10.5.2. To avoid bias due to OS competency, all shortcuts are disabled except for cut, copy, paste, find, and find-next, and participants are informed of these shortcuts. We also display the interface on a 24-inch wide-screen 1920-by-1200 resolution LCD monitor.

5.4 Programs

Each trial's error report was drawn from one of four open source programs, all of which we have previously applied Locksmith to: engine, aget, pfscan, and knot. We also chose reports in which imprecision in the sharing and lock state analysis do not contribute to the report's validity, so as to focus on the task of tracing program path as discussed in Section 2.

Reports from engine and pfscan are used during the tutorial and practice trials. Trials 1.1–1.3 use error reports from aget, and trials 2.1–2.3 use error reports from knot. By using different programs in each trial, we prevent participants from carrying over knowledge of the programs from the first interface to the second. The three selected reports within each program are significantly different from each other, e.g., they do not follow the same paths. This helps avoid bias that could arise if participants were given very similar triaging tasks for the same program.

Of the six reports, four contain 3 paths, and two contain 2 paths; eight paths have a call depth of 3, and the two deepest have a call depth of 8. Overall, there were 23 (non-*Locks*) tabs to complete for the experiment, 8 of which are true positives and 15 of which are false positives.

We also simplify the task slightly by making four small, semantics-preserving changes to the programs themselves. Doing so makes it simpler to ensure that our participants have a common knowledge base to work from, and reduces measurement variance due to individual differences. First, we made local static variables global. Second, we converted wait/signal synchronization to equivalent pthread_join synchronization when possible. We made both changes in response to confusion by some participants in our pilot study who were unfamiliar with these constructs. Third, we deleted lines of code disabled via #if 0 or other macros. Finally, in a few cases we converted gotos and switch statements to equivalent if statements. These last changes remove irrelevant complexity from the triaging task.

6. Experimental Results

6.1 Quantitative Results

Completion time We measured the time it took each participant to complete each trial, defined as the interval from loading the user interface until the participant submitted a completed checklist. The top part of Figure 7 lists the results, and the chart in the lower-left corner of the figure shows the mean completion times for each interface and session order combination. We found in general that PP results in 18% shorter completion times than SV.

More precisely, the mean completion time is 4:56 minutes for PP, and 6:02 minutes for SV. A standard way to test the significance

Completion times and accuracy for each trial

	Session 1				Session 2			
Trial	1.1	1.2	1.3	mean	2.1	2.2	2.3	mean
User		SV				PP		
1	8:36	14:14	9:44	10:51	7:07	4:48*	4:02	5:19
2	5:07*	3:10	5:50	4:42	4:16	2:29*	2:10*	2:58
5	7:46	2:34	5:38*	5:20	5:13	3:43*	1:18*	3:25
7	5:40	6:23	7:35	6:33	3:05	3:53*	2:32	3:10
				6:51				**3:43**
User		PP				SV		
0	6:27*	6:09	8:32*	7:03	9:42	5:16*	3:11*	6:03
3	6:38	7:18	8:35	7:30	11:18	6:21	3:39	7:06
4	8:21	2:11	4:43	5:05	5:26*	4:27*	2:46*	4:13
6	7:11	2:52	4:50**	4:57	4:33	4:06*	1:58*	3:32
				6:09				**5:14**
# Tabs	3	2	6		6	3	3	

* one incorrectly answered tab in the checklist

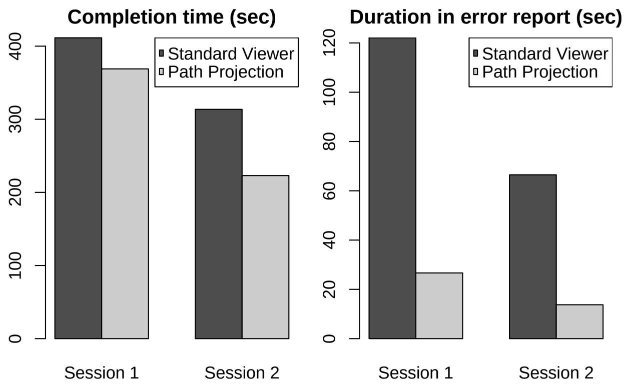


Figure 7. Quantitative data

of this result is to run a user interface (within subjects) by presentation order (between subjects) mixed ANOVA on the mean of the three completion times for each participants in each session.[3] However, this test revealed a significant interaction effect between the interface and the presentation order ($F(1, 6) = 6.046, p = 0.004$). (As is standard, we consider a p-value of less than 0.05 to indicate a statistically significant result.) We believe this is a learning effect—notice that for the SV-PP order (SV first, PP second), the mean time improved 3:08 minutes from the first session to the second, and for the PP-SV order the mean time improved 55 seconds. We ran two one-way, within-subjects ANOVAs, analyzing each presentation order separately, and we found that both of these improvements were statistically significant ($F(1, 3) = 12.78$, $p = 0.038$ and $F(1, 3) = 19.33, p = 0.022$, respectively).

However, notice that the SV-PP improvement is much greater than the PP-SV improvement. We applied Cohen's d, a standard mean difference test, which showed that the SV-PP improvement was large ($d = 1.276$), and the PP-SV improvement was small-to-medium ($d = 0.375$). This provides evidence that while there is a learning effect, PP still improves user performance significantly.

Note that when analyzing completion times, we did not distinguish correct and incorrect answers. Indeed, it is unclear how this affects timings, especially since in many cases, only one tab out of several will contain an incorrect answer.

Accuracy The chart at the top of Figure 7 also indicates, for each trial, how many of the checklist tabs were answered incorrectly. The total number of checklist tabs in each trial is listed at the bottom of the chart. We did not count the *Locks* tab in either of these numbers. User mistakes are evenly distributed across both interfaces. Participants made 10 mistakes (10.9%) under PP, compared to 9 mistakes (9.8%) under SV. For each participant, we summed the number of mistakes they made in each session, and we compared the sums for each interface using a Wilcoxon rank-sum test. This showed that the difference is not significant ($p = 0.770$), suggesting the distribution of errors is comparable for both interfaces. This shows that using PP, participants are able to come to similarly accurate conclusions in less time.

Most of the participant mistakes occurred in trials 2.2 and 2.3, in which two potentially-racing paths actually contain a common unrealizable sub-path, making the data race report a false positive. In trial 1.3, one participant completed two tabs incorrectly, but there was only one underlying mistake: misidentifying the same child thread as a parent thread (and thus affecting the answers to two tabs).

Time spent in error report We found that under PP, participants spend much less time with the mouse hovering over the error report compared to SV. The chart in the bottom right of Figure 7 shows the mean times for each session and interface. We found that on average, participants spend 20 seconds in the error report under PP, compared to 1:34 minutes under SV. This dramatic difference is highly significant (F(1,3)=15.634, $p < 0.001$) according to a user interface (within subjects) by presentation order (between subjects) mixed ANOVA.

We believe this difference is because PP makes the paths clearly visible in the source code display, whereas in SV, hyperlinks in the error report are heavily used to bring up relevant parts of the code. As additional evidence, participants themselves reported the same result: one noted that the error report is "necessary for the standard viewer, but just a convenience in [PP]."

Time spent in checklist Finally, we found that participants spend 27 seconds more on average with the mouse hovering over the checklist in PP compared to SV. Since checklist times across different trial numbers are incomparable, we applied a user interface-by-trial between-subjects ANOVA, which indicated a statistically significant result ($p = 0.007$).

We were surprised by this result, because we expected roughly equal times for this in both interfaces. In retrospect, however, we realized that participants need not scroll or click on the error report very often under PP, and hence they may leave the mouse cursor idle over the checklist.

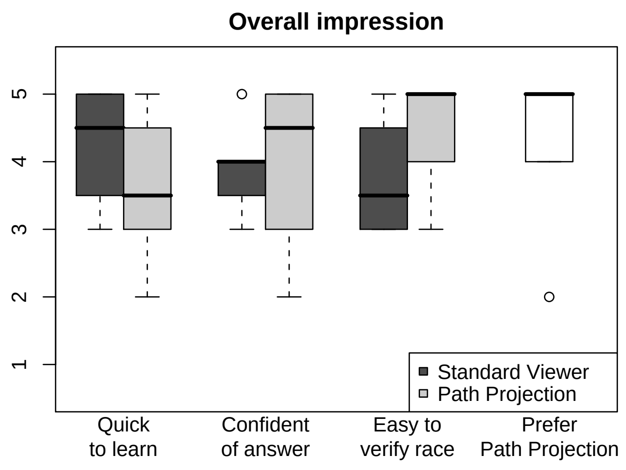
6.2 Qualitative Results

In addition to quantitative data, we also examined participants' answers to the questionnaires we administered. The questions are on a 5-point Likert scale, and we analyze them using either the Wilcoxon rank-sum test for paired comparisons, or the Wilcoxon signed-rank test otherwise.

Overall impressions The upper chart in Figure 8 gives a box plot summarizing participants' opinions of the interfaces. Comparing the median responses for the first three questions, we see that participants felt PP took longer to learn, led to more confidence in answers, and made it easier to verify a race. However, none of these results is statistically significant (p = 0.308, 0.629, 0.152, respectively), perhaps due to the small sample size ($N = 8$). When asked to compare the interfaces head-to-head, all but one participant preferred PP ($p = 0.016$).

However, we should also note that several participants felt that they were unable to fairly assess PP due to the novelty of the interface and the limited amount of exposure to it in the experiment. One participant rated PP worse than SV in all metrics, yet stated he preferred PP due to its potential, saying, "once you get comfortable

[3] We ran all statistical tests in R and SPSS. For ANOVA, we confirmed that all data sets satisfied normality using the Shapiro-Wilk test.



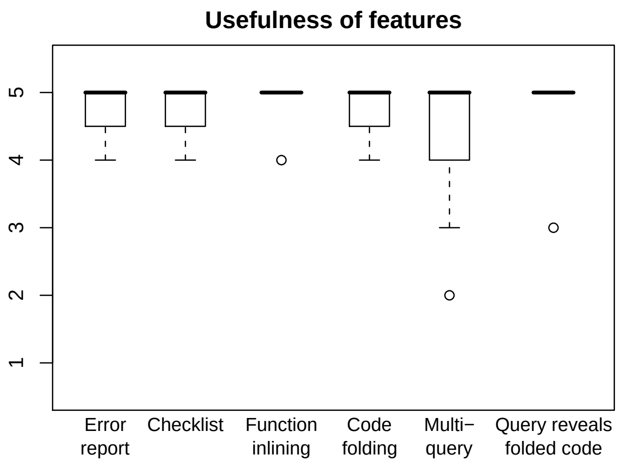


Figure 8. Participants' overall impression (1: disagree, 5: agree), and usefulness rating for particular features in Path Projection (1: not useful, 5: very useful)

with seeing [only] pieces of your code, I feel it will be more efficient."

Usefulness of features Finally, participants generally rated all the features of PP as somewhat or very useful. The bottom chart in Figure 8 shows a box plot of participants' responses, and all answers are statistically significant in favor of PP compared to a neutral response of 3 ($p < 0.032$).

While participants felt that the error report was very useful, they also commented that it served mostly as an initial overview under PP, whereas it was critical under SV. This matches our quantitative result indicating participants spent much less time in the error report under PP.

The checklist was very well received overall. One participant said, "[It] saved me from having to memorize rules." Interestingly, two participants felt that, while the checklist reduces mistakes, verifying the data race takes longer. This conflicts with our own experience—as we mentioned earlier, participants in our pilot study, which did not include the checklist, took notably longer to triage error reports than participants in our current study.

We thought that participants would be wary about function inlining and code folding, since the former is an unfamiliar visualization and the latter hides away much code. However, participants rated both very highly, saying particularly that code folding was "the best feature," "my favorite feature," and "I love using this feature [code folding]."

Most participants found multi-query useful, but two did not. When we asked them why, they replied that they felt that the code folding was already doing an adequate job without it. However, they had forgotten that the multi-query was in use by default with the four preset queries (shown in Figure 2). We believe that multi-query is actually extremely useful in combination with code folding.

6.3 Threats to Validity

There are a number of potential threats to the validity of our results. The core threat is whether our results generalize. We had a small number of participants, all of whom were students rather than expert programmers who had significant experience debugging data races. The set of programs ($N = 2$) and error reports ($N = 6$) in the experiment was small. Moreover, participants were asked to triage error reports for unfamiliar programs, rather than code bases they had developed themselves. Lastly, despite long tutorials before each experiment, we were unable to eliminate the learning effect across our trials.

However, even with these limitations, our experiment did produce statistically significant and anecdotally useful information for our population and sample trials. We leave carrying out a wider range of experiments to future work.

One minor threat is the small set of changes we made to the programs (Section 5.4) to avoid confusion in our participants. Using more expert programmers as test subjects would obviate the need for this. Also, the SV interface represents a typical IDE, but is not an actual, production-quality IDE. We chose this approach on purpose, so that familiarity with a particular IDE would not be a factor, but we may have omitted features that would be useful for our task.

7. Related Work

Error report triaging is essentially a program comprehension task. In our study, the particular task is to determine whether two dereferences could possibly execute simultaneously in two different threads. There is a substantial body of tools aimed at assisting in program comprehension tasks typically (but not exclusively) associated with code maintenance or re-engineering (e.g., SHriMP [40], and Code Surfer [1], to name just two more recent tools). As far as we are aware, none of these tools has been specifically designed to support users of static analysis for defect detection.

Conversely, many defect detection tools that use static analysis provide custom user interfaces [3, 30, 9, 26, 22, 1, 18, 8], IDE plugins [28, 19, 39, 23], or both [35]. However, we are not aware of any prior studies that measure the impact of the user interface on a static analysis tool's efficacy. Indeed, many proprietary tools have licenses that specifically prohibit publishing the results of studies about their tools.

Most of the above tools report the errors in part by using program paths, e.g., to illustrate a suspect data flow or an erroneous execution trace. We surveyed these tools based on publicly-available screenshots and literature and compared their features with Path Projection. As can be seen from Table 1, most interfaces share some features with Path Projection, though the implementation of these features is rarely the same. Code Sonar has many features in common with Path Projection, but we believe some of its interface choices could be improved. For example, several of Code Sonar's features insert additional colored lines of text into the source code. We find this results in a multitude of interleaving colored lines that make the source code hard to understand.

	Features (legend below)						
Tool	HLR	HPC	CF	FCI	AC	AQ	OV
Path Projection	×	×	×	×			
SDV [3]	×				×		
PREFix [30]		×			×		
PREFast [30]		×		*n/a*[1]	×		
Prevent [8]	×				×		
Code Sonar [22]	×	×	×	×	×		
MrSpidey [18]	×	×				×	×
Fortify SCA [19]	×		×				×
Klocwork [28]	×		×				
Fluid [39]	×		×				
FindBugs (GUI) [35]	×	×		*n/a*[1]			
FindBugs (Eclipse)	×		×	*n/a*[1]			
CQual [23]	×						

Feature legend

HLR : hyperlinked error report	AC : annotated code with analysis
HPC : highlighted path in code	AQ : analysis queries
CF : code folding	OV : other visualization (see text)
FCI : function-call inlining	

[1]PREFast and FindBugs are intraprocedural, so FCI does not apply.

Table 1. Summary of tool interface features

Several tools provide features not present in Path Projection, e.g., they insert summaries of the internal results of their analysis at corresponding lines in the source code. For example, SDV and Code Sonar visualize the state of an automaton-based specification at lines in the path. A few tools provide means to query an analysis' internal results at a given program line or expression. For example, MrSpidey allows the user to query possible run-time values for a given expression, based on a value-flow analysis. Some tools provide more graphical visualizations of paths. Fortify SCA illustrates a path as UML-style interaction diagram, and MrSpidey overlays lines on the source code to illustrate value flows. Path Projection's multi-query, path-derived code-folding, and check-list seem to be unique among the tools we studied. Interestingly, the FindBugs standalone GUI has features that are disjoint from its Eclipse plugin, which show how these features can sometimes be constrained by the IDE framework.

Triaging is a necessary first stage in identifying and fixing a software defect. In some organizations, there are programmers who are responsible solely for locating and documenting software defects into bug databases [37]. Work on bug triaging has been focused on detecting duplicate bug reports [36], assigning bug-fixing responsibility [2, 11], or visualizing the progress of bug reports [12]. Path Projection, and our study, differs from this work in exploring how triage can be performed more accurately and efficiently.

A number of proposals aim to improve the quality of error reporting by generating more intelligent error messages [15, 32, 4, 24, 42], or by prioritizing those messages [29]. This can be helpful, and our work complements these techniques by visualizing results in the context of code as much as possible.

8. Conclusions

We introduced Path Projection, a new program visualization toolkit that is specifically designed to help programmers navigate and understand program paths, a common output of many static analysis tools. Path Projection visualizes multiple paths side-by-side, using function call inlining and code folding to show those paths in a compact format. Our interface also includes a multi-query search facility to locate and reveal multiple terms simultaneously. To our knowledge, Path Projection is the first tool-independent framework for visualizing static analysis results in particular, and program paths in general.

We measured the performance of Path Projection for triaging error reports from Locksmith, a data race detection tool for C. For this experiment, we added a task-specific checklist that enumerates the sub-tasks needed to triage each error report. Our controlled user study showed that, compared to a standard viewer, Path Projection reduced completion times for triaging data races, with no adverse effect on accuracy, and that users felt that Path Projection was useful. We also found the checklist improved performance overall compared to a pilot study. To our knowledge, ours is the first study of the impact of the user interface on the efficiency and accuracy of triaging static analysis error reports.

We believe that Path Projection has many potential applications beyond bug triage. We intend to apply it to bug fixing and other programming tasks, such as interactive debugging. We also intend to study the use of checklists to complement static analysis tools.

9. Acknowledgments

We thank François Guimbretière for his advice and help in running our user study. We also thank Bill Pugh, Chris Hayden, Iulian Neamtiu, Brian Corcoran, and Polyvios Pratikakis for their suggestions and comments; and our pilot and actual study participants for their contributions to our research. This research was supported in part by National Science Foundation grants IIS-0613601, CCF-0430118, and CCF-0541036.

References

[1] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, July/August 2003.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06*, pages 361–370, 2006.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, Jan. 2002.

[4] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.*, 2(1-4):17–30, 1993.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18, Verona, Italy, Aug. 2004. Springer-Verlag.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207, 2003.

[7] S. K. Card, J. MacKinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan-Kaufman, 1999.

[8] Coverity, Inc. Coverity Prevent SQS, 2007. `http://www.coverity.com/html/prod_prevent.html`.

[9] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.

[10] Microsoft center for software excellence. `http://www.microsoft.com/windows/cse/default.mspx`.

[11] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE'04*, pages 92–97, 2004.

[12] M. D'Ambros, M. Lanza, and M. Pinzger. "A Bug's Life" Visualizing a Bug Database. *VISSOFT '07*, pages 113–120, 24-25 June 2007.

[13] D. Delmas and J. Souyris. Astrée: From research to industry. *Lecture Notes in Computer Science*, 4634:437–451, 2007.

[14] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI '07*, pages 435–445, 2007.

[15] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03*, pages 237–252, 2003.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI '00*, 2000.

[18] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *PLDI '96*, pages 23–32, 1996.

[19] Fortify Software Inc. Fortify Source Code Analysis, 2007. `http://www.fortifysoftware.com/products/sca/`.

[20] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, Nov. 2006.

[21] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, New York, NY, USA, 1986. ACM.

[22] GrammaTech, Inc. CodeSonar, 2007. `http://www.grammatech.com/products/codesonar/overview.html`.

[23] D. Greenfieldboyce and J. S. Foster. Visualizing Type Qualifier Inference with Eclipse. In *Workshop on Eclipse Technology eXchange*, Vancouver, British Columbia, Canada, Oct. 2004.

[24] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[25] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE '06*, pages 232–241, 2006.

[26] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *PLDI '02*, pages 69–82, 2002.

[27] C. C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the Value of Program Analysis Tools. In *OOPSLA'07 Practitioner Reports*, 2007.

[28] Klocwork Inc. Klocwork Enterprise Development Suite, 2007. `http://www.klocwork.com`.

[29] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *FSE '04*, pages 83–93, 2004.

[30] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, May/June 2004.

[31] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06*, pages 308–319, 2006.

[32] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP '03*, pages 15–26, 2003.

[33] J. Pincus. User Interaction Issues in Defect Detection Tools. Presentation at UW/MSR Research Summer Institute, 2001. `http://research.microsoft.com/users/jpincus/uwmsrsi01.ppt`.

[34] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, pages 320–331, 2006.

[35] B. Pugh et al. FindBugs, 2007. `http://findbugs.sourceforge.net`.

[36] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *ICSE '07*, pages 499–510, 2007.

[37] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings. In *ICSE '08*, 2008. To appear.

[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP '97*, pages 27–37, 1997.

[39] W. Scherlis et al. The Fluid Project, 2007. `http://www.fluid.cs.cmu.edu:8080/Fluid`.

[40] M.-A. Storey. *A Cognitive Framework For Describing and Evaluating Software Exploration Tools*. PhD thesis, Computing Science, Simon Fraser University, Canada, 1998.

[41] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[42] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting, 2000.