Prioritized Static Slicing for Effective Fault Localization in the Absence of Runtime Information

Yiji Zhang and Raul Santelices
University of Notre Dame, Indiana, USA
email: {yzhang20|rsantel}@nd.edu

ABSTRACT

Static slicing identifies the parts of a program that might affect another point in that program. Unfortunately, static slicing often produces large and imprecise results because of its conservative nature. Dynamic slicing can be a practical alternative, but it requires runtime information that might not be available, or be hard to obtain, or have low quality. To deal with the imprecision of static slicing, we present PRIOSLICE, a novel static approach that exploits the insight that not all statements reported by static slicing are equally likely to cause a program failure. PRIOSLICE first defines and solves a probabilistic model of program dependencies. In this model, some data dependencies are more likely to occur than others and control dependencies are less likely to propagate errors than data dependencies. PRIOSLICE then traverses the program backwards, just like static slicing, but prioritizing dependencies by likelihood. Our study of fault localization, presented in this paper, indicates that PRIOSLICE can help localize faults much more effectively than existing static-slicing approaches.

Keywords

program analysis, static slicing, probabilistic models, dependence analysis, fault localization

1. INTRODUCTION

Program slicing is a popular program-analysis technique used for various software-quality tasks, such as fault localization and program comprehension [39]. Static program slicing, in particular, finds the set of all statements in the program—the slice—that might affect a particular point in that program. Unfortunately, however, static slicing is often too imprecise to be practical because it tends to produce slices whose sizes are very large [9,32], even when considering calling contexts for interprocedural analysis [20].

To reduce the size of slices and, thus, increase their usefulness, researchers have developed other forms of program slicing such as dynamic slicing [3,24], variants of it [8,13,18], combinations of static slices with execution data [17,19,25,27], and slice pruning based on some criteria [2,11,36,41]. These variants, however, focus only on subsets of all program behaviors and, thus, can miss faulty code. Moreover, these techniques can still be imprecise [32].

Dynamic approaches provide concrete insights on how programs behave in typical cases, provided that a “representative” set of executions is used. Unfortunately, such a set of executions is not always available to developers. For fault localization, one failed execution can be enough to find a fault, but multiple executions are often needed to make localization effective [22,23]. In many cases, such as bugs reported by users, developers may not even have executions to work with. Another problem is that faulty executions can be hard to reproduce for debugging. This problem is exacerbated by non-determinism in software whose behavior depends on external factors such as time and thread interleavings. One possibility is to use and deploy software instrumented to collect dynamic slices, but for unacceptable runtime overheads. Therefore, dynamic slicing is not always an option and static slicing is often needed.

In this paper, we present a novel technique called PRIOSLICE which considerably increases the effectiveness of static slicing by reducing the negative effects of its imprecision. PRIOSLICE prioritizes the inspection of a static slice (i.e., the subset of the program that affects a particular program point) using a probabilistic model of how dependencies [4,15]—the building blocks of program slices—occur and how they affect the slicing criterion (i.e., the value in the program point from which the slice is computed). To that end, for each statement in a slice, PRIOSLICE computes a weight in the range [0,1] representing the likelihood that this statement belongs to the slice. Thus, static slicing with PRIOSLICE tells not only whether a statement is in a slice, but also how much.

The probabilistic model of PRIOSLICE exploits two key insights. The first insight is that not all data dependencies [4] are equally likely to occur because of control-flow and aliasing [5] reasons. Data dependencies are treated (mostly) as equals by existing static-slicing techniques [36,39]. Our model and technique, in contrast, gives a greater priority to the data dependencies that are most likely to occur and to cause a particular program behavior (e.g., a failure) and, thus, more quickly identify its causes (e.g., a fault).

The second insight used in this model is that control dependencies [15] are, in general, weaker than data dependencies at propagating the effects of faults [26,32,36]. Therefore, they should not be treated as equals with data dependencies for slice inspection—as Weiser’s method does [39]—but they should not be discarded upfront either—as, for example, other techniques such as thin slicing do [36]. Our model incorporates both types of dependencies while giving them different weights according to their respective fault-propagation potential as estimated by our probabilistic analysis.

Using a dependence analyzer and this model, PRIOSLICE first computes the static backward slice from a slicing criterion in a program and then solves the system of equations given by the model
for the dependence graph of that program. The solution for these equations is the set of weights for the statements in the static slice. These weights are obtained using existing approaches for solving real-valued data-flow problems [29]. PRIOSLICE uses these results to perform a best-first traversal [28] of the slice, which modifies Weiser’s breadth-first traversal approach [39] of static dependencies by prioritizing statements according to their weight.

To study the feasibility and effectiveness of PRIOSLICE for fault localization, we implemented this technique for slicing Java byte-code programs based on our dependence-analysis infrastructure [30, 34]. In this study, presented in this paper, we applied and compared three static-slicing techniques—PRIOSLICE, Weiser’s traversal approach for static slicing, and thin slicing—to various faults in “real-world” Java subjects: NanoXML, XML-security, JMeter, and Jaba.

Our results for these subjects and faults indicate, with statistical significance, that PRIOSLICE can reach the faulty code faster than static slicing. PRIOSLICE traverses, on average, about 17% of the program, whereas Weiser’s approach requires an average inspection of nearly 27% of the program to find each fault. Meanwhile, thin slicing—when applicable—needs 26% of the program.\(^1\) For the faults for which thin slicing is applicable, PRIOSLICE requires only about 14% of the program to be inspected.

The most important benefit of this work is that it provides a new way of looking at static slices for fault localization, in which statements are distinguished by relevance rather than just by membership in the slice. Moreover, PRIOSLICE has the potential to be used for many other applications, including program comprehension and speculative parallelization. For instance, we have used a preliminary forward version of this model [31, 35] for change-impact analysis. Also, our probabilistic model has plenty of room for improvement in assigning weights with greater precision and, thus, greater effectiveness.

In all, the contributions of this paper include:

- A new form of static backward program slicing that indicates not only whether, but also how much, each statement belongs to the slice.
- A new technique, PRIOSLICE that realizes this idea of probabilistic slice to prioritize the traversal of slices by the estimated relevance of their statements.
- A study indicating that PRIOSLICE can considerably reduce the effort of static fault localization with respect to Weiser’s original slicing approach and thin slicing.

2. BACKGROUND

This section presents core concepts necessary for understanding the rest of the paper and illustrates these concepts using the example program in Figure 3.

2.1 Program Dependencies

Control and data dependences are the building blocks of program slicing [20, 39]. A statement \(T\) is control dependent [15] on a statement \(S\) if a branching decision taken at \(S\) determines whether \(T\) is necessarily executed. A control dependence \((S, T)\) occurs at runtime (i.e., it is covered) if the source \(S\) of the control dependence is reached and a branch is taken from \(S\) that necessarily causes \(T\) to execute. For example, line 4 is control dependent on line 3 in the example program of Figure 3 because the decision made at line 3 determines whether line 4 is executed or not.

A statement \(U\) is data dependent [4] on a statement \(D\) if a variable \(v\) can be defined (written to) at \(D\) and used (read from) at \(U\) and there is a definition-clear path in the program for \(v\) (i.e., a path that does not re-define \(v\)) from \(D\) to \(U\). A data dependence \((D, U, v)\) or, simply, \((D, U)\) occurs at runtime if the definition is reached, the memory location \(v\) is written in that definition, a definition-clear path for \(v\) to use \(U\) is then traversed, and \(v\) is in fact the location read in that use. (If the variable is accessed via a pointer or reference, it might be possible that the memory location written to is not the same read from.)

2.2 Program Slicing

Program slicing was originally developed by Weiser [39]. A static slice for a set of variables \(V\) at a program point \(C\)—the slicing criterion—is the subset of all program statements that affect those values at that point. Weiser’s approach performs a backward breadth-first traversal of the data and control dependencies in the program from the slicing criterion. This traversal produces a visit order of the statements that can be used to measure the effort required to find a fault as the location of the fault in that order.

To illustrate, consider the example program in Figure 3. Suppose that we want to determine all statements in this program that affect the value of \(v\) at statement 16. Using the backward transitive closure of control and data dependencies from that statement, we obtain the slice \(\{1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13\}\).

3. DATA DEPENDENCE MODEL

In preliminary work [42], we proposed a novel technique called PRIODU to estimate the execution likelihood of all data dependencies in a program. Our ultimate goal of prioritizing slice inspections, stated in that work, is realized now in this paper with PRIOSLICE. For PRIODU, we used control-flow and points-to analysis to assign a score for each data dependence which represents its likelihood of occurrence, and ranked all data dependencies by the decreasing order of the scores. Our insight was that not all data dependencies are equally likely to occur at runtime. To further leverage previous findings and help developers in fault localization, now we expand PRIODU to all types of dependencies—control and data dependencies—and combine it with static backward slicing information to form another system of equations. In this section, we introduce the aspects of PRIODU that we use for data dependencies in our new technique, PRIOSLICE.

3.1 Reaching Probability of Statements

A key part of PRIODU is the concept of reaching probability. For statements \(a\) and \(b\), the event \(a \rightarrow_R b\) states that the program follows some path from \(a\) to \(b\). The probability of \(a \rightarrow_R b\) is:

\[
P(a \rightarrow_R b) = \begin{cases} 1 & \text{if } a = b; \\ P(a \rightarrow_R b \mid a \neq b) & \text{otherwise} \end{cases}
\]

(1)

If \(a\) and \(b\) are the same statement, the reaching probability is trivially 1. Otherwise, if \(a\) is different from \(b\), \(P(a \rightarrow_R b)\) is the probability of reaching \(b\) from \(a\) through some successor of \(a\):

\[
P(a \rightarrow_R b \mid a \neq b) = \frac{\bigvee_{s \in \text{succ}(a)} a \rightarrow s \land s \rightarrow_R b}{|\text{succ}(a)|}
\]

\[
= \sum_{i = 1}^{\text{|succ}(a)|} P(a \rightarrow_R s_i) \times P(s_i \rightarrow_R b)
\]

(2)
where \( \text{succ}(a) \) is the set of all control-flow successors of statement \( a \) and \( a \rightarrow_S s \) states that \( s \) is the successor that executes right after \( a \). This probability is the sum of the probabilities of reaching \( b \) through the successors of \( s \) because the events \( a \rightarrow_S s \land s \rightarrow_R b \) in the disjunction are exclusive—exactly one statement succeeds another in an execution. Also, the two events in each term of the disjunction are independent, so their joint probability is the product of their respective probabilities. Note that the probability of the second event, \( s_i \rightarrow_R b \), can make Equation 2 recursive. Thus, approaches for solving real-valued data-flow problems must be used [29]. The probability of \( a \rightarrow_S s_i \), specifically, is given by a branch prediction mechanism (a parameter of our technique). The only constraint is

\[
\sum_{i=1}^{\text{succ}(a)} P(a \rightarrow_S s_i) = 1 \tag{3}
\]

### 3.2 Impact Probability of Data Dependencies

Another building block of PRIODU that we use in PRIOSLICE is the impact probability of data dependencies. A data dependence \((d, u)\) propagates an impact if (1) \( u \) is a reachable use of \( d \) and there is a definition-clear path from \( d \) to \( u \), and (2) the memory location defined at \( d \) and used at \( u \) is the same. We define these two events as \( d \rightarrow_R u \) and \( \text{alias}(d, u) \), respectively. Our model defines the impact probability of \((d, u)\) as

\[
P(d \rightarrow_R u) = P(d \rightarrow_R u \land \text{alias}(d, u)) = P(d \rightarrow_R u) \times P(\text{alias}(d, u)) \tag{4}
\]

where the alias event is possibly dependent on the choice of path from \( d \) to \( u \). However, computing the probability of these two events as dependent events is impractical because of the large or even infinite number of paths from \( d \) to \( u \). For this reason, to simplify our computations, we treat the two events as independent and multiply their respective probabilities.

The probability of the first event on the right-hand side of Equation 4 is computed using the following formula:

\[
P(d \rightarrow_R u) = \begin{cases} 
0 & \text{if } u \notin \text{succ}(d); \\
1 & \text{if } u \in \text{succ}(d); \\
P \left( \bigvee_{s \in \text{succ}(d)} \left( d \rightarrow_S s \land u \in \text{ru}(s) \right) \wedge s \rightarrow_R u \right) & \text{otherwise}
\end{cases} \tag{5}
\]

where \( \text{ru}(s) \) is the set of reachable uses [4] from statement \( s \). Equation 5 states that if the use \( u \) is an immediate successor of the definition \( d \), the probability for the data dependence is the probability that \( u \) succeeds \( d \). Otherwise, the probability for the dependence is that of reaching \( u \) through some successor \( s \) via a definition-clear path. The three events \( d \rightarrow_S s, s \in \text{ru}(s), \) and \( s \rightarrow_R u \) are independent. Hence, this probability can be decomposed into a sum of the products of the probabilities of these events, in a similar fashion to Equation 2. The third event makes the computation of the impact probability for a data dependence also a recursive problem.

The second term of Equation 5 is the event that the defined and used memory locations are the same at runtime. Our technique uses points-to sets [5] to describe memory locations defined and used. It estimates the aliasing probability as the probability that any element from one points-to set is also in the other set, which corresponds to the Jaccard similarity of those sets [21].

Figure 1 shows the big picture of how PRIOSLICE works. First, a developer uses a program \( P \) with test suite \( T \) on her own computer and a failure occurs. The user then submits a bug report describing the experienced failure. A developer interprets the bug report and determines the failure location in the program. Then, PRIOSLICE computes, for each dependence \((s, t)\) in \( G \), the impact probability that the source \( s \) propagates a fault to the target \( t \) via this dependence. Then, PRIOSLICE performs a prioritized traversal of \( G \) using the impact probabilities of its dependencies to produce a visit order of the statements for the developer. Statements that have the same probability are inspected together or in some arbitrary order.

Figure 3 shows an example program that helps us illustrate our technique. The program takes five integers as inputs and calls, up to four times, the helper function \text{printSqrt}, depending on control-flow decisions influenced by the input values. \text{printSqrt} takes an integer argument and checks whether that argument is greater or equal than zero (a precondition for computing a square root), calculates the square root of the argument, and then prints this square root. The example contains a fault (bug) in line 9: when the input \( n \) is less than the variable \( m \), the program proceeds to line 10 to assign \( m \) a possibly negative value of \( m+n \), and when \( n \) is greater than \( m \), line 11 definitely assigns \( m \) a negative value. Thus, \text{printSqrt} can be called at lines 10, 11, or even 13 with a negative argument that makes the program fail at line 16 by violating the assertion.

The right part of Figure 3 shows the dependence graph for this example program to help visualize how line 16 is affected by the rest of the program in general and by the fault at line 9 in particular. Each node in the graph represents a line (statement) in the

\[
P(\text{alias}(d, u)) = \frac{|P^2 \text{Set}(d) \cap P^2 \text{Set}(u)|}{|P^2 \text{Set}(d) \cup P^2 \text{Set}(u)|} \tag{6}
\]
program\textsuperscript{2} and each edge represents a data dependence (solid edge) or control dependence (dotted edge) between two statements. In this graph, the shaded nodes correspond to the statements in the static backward slice from value $v$ at line 16.

After slicing the program, P\textsc{rio} solves a system of equations, described later in Section 4.2, to estimate the probability of each statement in the slice of affecting the slicing criterion. Intuitively, P\textsc{rio} works as follows: assuming that the developer has reached a line $t$ during inspection, line $s$ affects line $t$ to the extent (likelihood) that (1) line $s$ is also reached in the program and (2) a faulty state that reaches $s$ or originates at $s$ propagates to $t$ via a sequence of dependencies (data, control, or both).

For the first event, P\textsc{rio} performs a reaching-probability analysis to estimate the likelihood that a statement is reached from the entry of the program for any potential execution. For example, in Figure 3, line 15 is reached if any of the conditions at lines 3, 6, 10, 11, and 12 evaluates to true. The probability of reaching line 15 is 1 because either outcome of the condition at line 9 causes the helper function to be called. Each of the call sites for that function, however, has a reaching probability corresponding to the probability that the guarding conditional takes the branch leading to that call site (e.g., 0.5).

In general, there are multiple control-flow paths to a target statement in a program and the possible outcomes of each conditional statement are exclusive events. For those cases, we use the property that the probability of the disjunction of independent events is the sum of terms $i$ in which each term $i$ is the product of the probability of event $i$ and the probability that all terms $j (j < i)$ do not occur.

For the second event, P\textsc{rio} first estimates, for each dependence, how likely it is that the dependence is covered (executed) and that it propagates a fault from its source to its target. In Figure 3, for example, the probability of covering data dependence (1,9) is 1 simply because line 9 always executes after line 1 (assuming no violations of the assertion at line 16 during intermediate calls to print\textsc{sqrt}). Also for this dependence, with probability 1, the variable $v$ defined at the source line is the same used at the target line because it is accessed directly and not via pointers or references (i.e., the probability of aliasing for this variable is 1). Therefore, the propagation probability of dependence (1,9) is $1 \times 1 = 1$. For a less trivial example, P\textsc{rio} estimates that the probability that line 1 affects line 10 via data dependence (1,10) is the probability of executing branch (9,10) (e.g., 0.5).

For a control dependence, the coverage probability is simply the probability that the corresponding branch or control-flow edge is taken. As for propagation, if the source of the control dependence is faulty or a fault reaches that source, the fault propagates to the target of the dependence only if another branch from the source should have been taken instead. In other words, a propagation to the target implies that the branch was taken by mistake. To illustrate, consider control dependence (9,10) in Figure 3. If, for example, the probability of the corresponding branch is 0.4, then the probability that a fault causes the program to wrongly take this branch is $1 - 0.4 = 0.6$. Hence, the impact probability of (9,10) is the product of covering this branch by mistake: $0.4 \times 0.6 = 0.24$.

For statements $s$ on which the failing statement $f$ is indirectly dependent (i.e., the dependence graph paths from $s$ to $f$ are longer than 1), each statement $s$ affects the failure $f$ through one or more successors of $s$. Consider, for example, line 9 in Figure 3. The probability of this line affecting line 15 via the call at line 13 has three parts: line 9 affects line 10, line 10 affects line 13, and line 13 affects line 15. The probability of these factors are, respectively, 0.24, 0.5, and 1, for a total probability of 0.12. More generally, there can be many control-flow paths along which one line affects another and this computation can be recursive because of loops.

As an example of a fault that can propagate through multiple dependence-successors, consider line 1 in Figure 3. Line 1 has an impact on line 15 if it has an impact through any of its successor lines 3, 6, 9, 10, 11, and 12. Note that lines 10 and 11 are also

---

\textsuperscript{2}For simplicity, we omitted the customary entry and start nodes.
impacted indirectly by line 1 via line 9. PRIOSLICE takes into account all impacts, direct or indirect, of a line on any another line. The paths for some of those impacts can overlap. In this example, all control-flow paths cover the dependence paths of line 15 on 1.

Because dependence paths can overlap, they might not be independent events from a probabilistic perspective. However, it is impractical to compute the probabilistic dependence among those paths. Hence, PRIOSLICE uses Fréchet inequalities [10] for probability bounds of disjunctions of events whose dependence is unknown. The lower bound is the minimum of all probabilities and the upper bound is the sum of the probabilities or 1 if the sum is greater than 1. PRIOSLICE assigns the middle value of the lower and upper bounds to these disjunctions. For example, if the calls to printSqrt at lines 10 and 11 did not exist, and if the impact probabilities of line 1 on line 15 via lines 4, 7, and 13 are all 0.5, the overall impact probability is \((0.5 + \min(0.5+0.5+0.5,1))/2 = 0.75\).

After computing the impact probabilities for all dependencies, PRIOSLICE performs a backward best-first traversal\(^3\) [28] of the dependence graph from the slicing criterion using these probabilities. To illustrate, Table 1 shows the traversal steps from line 16 in our example. The first column shows the visit order and the second column lists the candidate statements (discovered but not yet visited) and their impact probabilities. In each step, PRIOSLICE picks the candidate (or set of candidates, if there is a tie at the top) with the largest probability of impacting the failure and adds it (or them) to the prioritized slice in the next row. When visiting a statement, the predecessors are added to the candidate list for the next step. For this example, the table shows that PRIOSLICE traverses only 5 statements to locate the faulty statement 9.

We also want to compare PRIOSLICE with Weiser’s traversal of static slicing and with thin slicing. Weiser’s slicing performs a breadth-first traversal from statement 16. Statement 16 is at depth 1 and the fault (statement 9) is at depth 6 in this traversal. There are 10 nodes in this search before reaching depth 5 and there are three nodes at depth 5. Thus, the effort of locating the fault is \((11+13)/2=12\), which is considerably greater than using PRIOSLICE. Meanwhile, thin slicing does not discover this fault because all dependencies from the faulty statement 9 are control dependencies, separating the cases of data and control dependencies: to obtain the impact probability of data dependencies presented in Section 3 to both types of dependencies. These probabilities are necessary because a fault in a program propagates in the program through some sequence of dependencies that can combine both data and control dependencies. To obtain the impact probability of a dependence, PRIOSLICE estimates how likely it is to cover (execute) the dependence and how likely it is that, when covered, the dependence also has an impact by propagating a fault that reaches (or is created at) its source statement to its target statement.

The following notation describes the impact probability of a dependence, separating the cases of data and control dependencies:

\[
P(a \rightarrow_D b) = \begin{cases} 
P(d \rightarrow dd_{IJ} u) & \text{if } D \text{ is a data dependence;} 

P(s \rightarrow cd_{IJ} t) & \text{if } D \text{ is a control dependence} 
\end{cases} 
\]  

(7)

where \(a \rightarrow_D b\) indicates that \((a,b)\) is a dependence of statement \(b\) on \(a\) and that an impact occurs through this dependence—a fault propagates via \((a,b)\). The marks \(dd\) and \(cd\) replace \(D\) to specify data and control dependence, respectively. Our model defines how to compute the impact probability for each of the two types.

Like data dependencies, a control dependence \((a,b)\) has an impact if and only if it is covered and a faulty state propagates from the source to the target statement. For this type of dependence, the coverage probability is that of covering the corresponding control-flow edge \(e\) and the impact probability is the probability that \(e\) is the wrong edge to take (i.e., the probability that, without the fault, a different successor of \(a\) would have been taken). Thus, the impact probability of a control dependence \((a,b)\) is

\[
P(a \rightarrow_C cd_{IJ} b) = P(a \rightarrow s b) \times (1 - P(a \rightarrow s b))
\]  

(8)

4.2 Probabilistic Model

In addition to data dependencies (Section 3), we also need to model control dependencies as a building block of PRIOSLICE. Here, we expand the impact probability of data dependencies presented in Section 3 to both types of dependencies. These probabilities are necessary because a fault in a program propagates in the program through some sequence of dependencies that can combine both data and control dependencies. To obtain the impact probability of a dependence, PRIOSLICE estimates how likely it is to cover (execute) the dependence and how likely it is that, when covered, the dependence also has an impact by propagating a fault that reaches (or is created at) its source statement to its target statement.

The following notation describes the impact probability of a dependence, separating the cases of data and control dependencies:

\[
P(a \rightarrow_D b) = \begin{cases} 
P(d \rightarrow dd_{IJ} u) & \text{if } D \text{ is a data dependence;} 

P(s \rightarrow cd_{IJ} t) & \text{if } D \text{ is a control dependence} 
\end{cases} 
\]  

(7)

where \(a \rightarrow_D b\) indicates that \((a,b)\) is a dependence of statement \(b\) on \(a\) and that an impact occurs through this dependence—a fault propagates via \((a,b)\). The marks \(dd\) and \(cd\) replace \(D\) to specify data and control dependence, respectively. Our model defines how to compute the impact probability for each of the two types.

Like data dependencies, a control dependence \((a,b)\) has an impact if and only if it is covered and a faulty state propagates from the source to the target statement. For this type of dependence, the coverage probability is that of covering the corresponding control-flow edge \(e\) and the impact probability is the probability that \(e\) is the wrong edge to take (i.e., the probability that, without the fault, a different successor of \(a\) would have been taken). Thus, the impact probability of a control dependence \((a,b)\) is

\[
P(a \rightarrow_C cd_{IJ} b) = P(a \rightarrow s b) \times (1 - P(a \rightarrow s b))
\]  

(8)

---

\(^3\)This can be seen as generalized breadth-first search where nodes to visit are picked by priority first, even if they are deeper in the graph.
PRIOSLICE prioritizes the inspection of statements by traversing the static dependence graph backward according to the impact probabilities of the dependencies already traversed. During the traversal, if the user is inspecting a statement \( b \) it is because that user has reached \( b \) through a sequence of dependencies from the criterion \( C \). Also, PRIOSLICE weighs dependencies by the likelihood of reaching their source statements from the entry of the program. Therefore, the impact probability that we need for a dependence \((a,b)\) during traversal must include the reaching of \( a \) from the entry and must be conditional on having reached the target \( b \):

\[
P_{PRIOSLICE}(a \xrightarrow{D} b) = P(E\xrightarrow{RA} b \land a \xrightarrow{D} b \mid E\xrightarrow{Rb}) = \frac{P(E\xrightarrow{RA} b \land a \xrightarrow{D} b \land E\xrightarrow{Rb})}{P(E\xrightarrow{Rb})} = \frac{P(E\xrightarrow{RA} b) \times P(a \xrightarrow{D} b \mid E\xrightarrow{Rb})}{P(E\xrightarrow{Rb})} \tag{9}
\]

where \( E\) is the entry point of the program and \( E\xrightarrow{RA} a \) and \( a \xrightarrow{D} b \) are independent from each other. Also, these two events imply \( E\xrightarrow{Rb} \). Thus, the conditional probability of a dependence for the PRIOSLICE traversal is the product of the probabilities of reaching its source and having an impact on its target divided by the probability that the target has been reached.

Concretely, PRIOSLICE performs a best-first backward traversal [28] of the static dependence graph from the slicing criterion using the conditional impact probability of each statement as its priority. In this traversal, after visiting a node, the predecessors are discovered: their priorities are calculated (or re-calculated, if already discovered) and they are added to a priority queue. This queue contains all neighbor nodes in the graph that are candidates for the next visit. The node that is visited next and popped from the queue is the one with the highest priority in that queue.

To calculate or update the impact probability of a statement \( a \) on the slicing criterion \( C \) at any moment during the traversal with the information available so far in that traversal, PRIOSLICE goes through all dependencies \((a,s)\) such that \( s \) has been already visited or discovered, using this formula:

\[
P(a \xrightarrow{s} C) = \begin{cases} 
1 & \text{if } a = C; \\
\frac{1}{P \left( \bigvee_{s \in \text{suc}(a)} s \xrightarrow{s} C \right)} & \text{otherwise} \tag{10}
\end{cases}
\]

where \( a \xrightarrow{s} C \) is the event that \( a \) impacts \( C \) via dependence-successor \( s \) and where \( \text{suc}(a) \) depends on the variant of PRIOSLICE. For PRIOSLICE 1 and 3, \( \text{suc}(a) \) is the subset of dependence-successors of \( a \) that have been visited (i.e., discovered nodes already pulled out of the priority queue). For PRIOSLICE 2, \( \text{suc}(a) \) consists of all discovered nodes (i.e., either visited or still in the queue), which provides more data but at a greater cost.

For a dependence \((a,b)\), if \( a \notin C \), \( a \) impacts \( C \) via \( b \) if and only if \( a \) is reached at runtime, \( a \) impacts \( t \) via this dependence, and \( b \) impacts \( C \). Therefore, the probability that \( a \) has an impact on the slicing criterion through a dependence successor \( b \) is

\[
P(a \xrightarrow{b} C) = P_{PRIOSLICE}(a \xrightarrow{D} b) \times P(b \xrightarrow{} C) \tag{11}
\]

which makes Equation 10 potentially recursive.

The terms in the disjunction in Equation 10, however, are not necessarily exclusive. One path from \( a \) to \( b \) might be probabilistically dependent on another such path. However, it is impractical to compute the exact dependence relationships among those paths. Therefore, we estimate the disjunction of events whose dependence is unknown using Fréchet inequalities [10]:

\[
\max \{ P(A_i) \} \leq P \left( \bigvee_{i=1}^{n} A_i \right) \leq \min \left( 1, \sum_{i=1}^{n} A_i \right) \tag{12}
\]

Fréchet inequalities give the lower bound and upper bound of the probability of a disjunction of events. Without assuming any dependence among events, we estimate the probability as the middle point of the lower and upper bounds:

\[
P \left( \bigvee_{i=1}^{n} A_i \right) = \frac{\max \{ P(A_i) \} + \min \left( 1, \sum_{i=1}^{n} A_i \right)}{2} \tag{13}
\]

### 4.3 Algorithm

Figure 4 shows the PRIOSLICE algorithm that traverses the dependence graph of a program backward from a slicing criterion using a best-first strategy. The algorithm uses the equations presented in Sections 4.2 and 3 and takes the three variants of PRIOSLICE into account, according to the parameter \( T \), via lines 15 and 16.

Line 1 computes the static dependence graph of the program from the slicing criterion. Lines 2–6 initialize two sets of probabilities: the reaching probability from the program entry for each statement in the graph and the conditional impact probability from the entry of the program for each dependence. Lines 7–9 initializes the data structures for the priority queue and the outputs (ranking list \( R \) and probability map \( M \)) by seeding them with the slicing criterion and impact probability 1.

The main loop in lines 10–19 proceeds until no new statements are left to process in the priority queue (i.e., the graph has been completely traversed) or the user decides to stop for some reason (e.g., a fault was found). Line 11 picks for visit and removes the
next statement \( n \) from the priority queue (the statement with the highest priority) and line 12 adds it to the ranking list if not already there. The visit of \( n \) continues with lines 14–18 which, if the conditions for the technique version allow (line 15), compute the impact probabilities of the predecessors \( m \) of \( n \) and adds (or updates) both \( m \) and its probability to the priority queue and the output map.

5. **EMPIRICAL EVALUATION**

To assess the effectiveness of PriorSLICE for localizing faults in absolute terms and in comparison with slicing, we performed an empirical evaluation on 28 faults across four “real-world” programs. Specifically, our goal was to find out how much of a program must be inspected by PriorSLICE and two static-slicing approaches: Weiser’s traversal approach of static slices and thin slicing [36] (an alternative that focuses on data dependencies). Our research questions are:

- **RQ1:** How effective is PriorSLICE for fault localization compared to other slicing techniques?
- **RQ2:** How practical is PriorSLICE for fault localization in terms of its running costs?

The effectiveness of a fault-localization technique is inversely proportional to the effort a user needs to locate the fault using the corresponding dependence-graph visit order. The practicality is inversely proportional to the computational costs of the technique.

5.1 **Experimental Setup**

We implemented PriorSLICE in Java as a new module of DUA-FORENSICS, our dependence analysis and instrumentation toolset [30, 34]. Our implementation for this study inputs a Java-byte-code program, a failure location, and a fault to locate. It computes the static dependence graph from the failure point (slicing criterion), determines the traversal orders for PriorSLICE, static slicing, and thin slicing, and determines the effort for locating the fault using each order. For simplicity, we assigned to all control-flow edges \( E \) from the same statement the probability \( 1/E \). As our results show next, this assumption was sufficient to demonstrate the effectiveness of PriorSLICE.

Table 3 lists our subjects, their size in lines of code (LOC), and the number of faults we obtained for each subject. NanoXML, XML-Security, and JMeter are provided by the SIR repository [14] along with faults seeded by other researchers and test suites that expose these faults, which we used to identify the failure points. Jaba is provided upon request by the Aristotle Research Group from Georgia Tech, including real faults found by its developers (not us) and a test suite.

The criterion for selecting a fault for our study is that the faulty statement must be covered by the test case and that there must be at least one test case that exposes the bug by means of a different output of that test case when the fault is inserted. Table 4 lists the ids of the faults (as provided with those subjects) that satisfied our criterion for the study. For each of the three largest subjects, XML-Security, JMeter, and Jaba, only seven bugs are eligible. For our criterion for the study. For each of the three largest subjects, XML-Security, JMeter, and Jaba, only seven bugs are eligible. For Jaba, we included all dependencies, whereas for thin slicing we omitted dependencies where the variable at the use is a reference to an object that has been destroyed before reaching that statement during the traversal of the corresponding dependence graph (with the appropriate omissions for thin slicing). We implemented thin slicing without the iterative addition of initially omitted dependencies mentioned by its authors because there is no clear order in which those dependencies are added next [36].

For Weiser’s static slicing and for thin slicing, we used breadth-first backtracking of the graphs to determine the order in which the statements in the slices are inspected. For Weiser’s slicing, we included all dependencies, whereas for thin slicing we omitted control dependencies and base-variable data dependencies (i.e., dependencies where the variable at the use is a field of a method of the object to which the method belongs) [36].

We implemented thin slicing without the iterative addition of initially omitted dependencies mentioned by its authors because there is no clear order in which those dependencies are added next [36].

<table>
<thead>
<tr>
<th>subject</th>
<th>description</th>
<th>LOC</th>
<th># of faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>NanoXML</td>
<td>XML parser</td>
<td>3521</td>
<td>?</td>
</tr>
<tr>
<td>XML-Security</td>
<td>encryption library</td>
<td>22361</td>
<td>?</td>
</tr>
<tr>
<td>JMeter</td>
<td>performance tester</td>
<td>35547</td>
<td>?</td>
</tr>
<tr>
<td>Jaba</td>
<td>program analyzer</td>
<td>37920</td>
<td>?</td>
</tr>
</tbody>
</table>

5.2 **Methodology**

Our experimental process used the following steps for each fault:

- **1. Identify the failing point.** To locate the failure, we took the first test case for the subject that fails for the given fault and identified the first statement involved in that failure. The failing statement is either the one that prints the first wrong value, the statement that throws an uncaught exception, or the conditional statement that skips the printing of the first missing output value. This step is a practice of the process in Figure 1 from the user running the program to obtaining the failure statement.

- **2. Obtain the dependence graph.** We used DUA-FORENSICS with this failing point as the slicing criterion to obtain the portion of the static dependence graph that affects that point. This process is corresponding to the first step in Figure 2.

- **3. Apply each technique.** We applied each traversal technique—PriorSLICE, Weiser’s static slicing, and thin slicing—and measured the respective efforts, as fractions of the program, required to reach (localize) the first faulty statement \( f \) using this formula:

\[
\text{effort}(f) = \frac{\text{rank}(f)}{\text{program size}}
\]

where \( \text{rank}(f) \) is the number of statements that had to be visited to reach \( f \) and \( \text{program size} \) is the total number of statements in the program. In case of a tie—when the visit priority of \( f \) is the same as for other statements—\( \text{rank}(n) \) is the average position of \( f \) and all statements tied with it. For example, if four statements are tied at positions 3 to 6, their rank is \((3+6)/2 = 4.5\). For Weiser’s static slicing and for thin slicing, we used breadth-first backtrafsals of the graphs to determine the order in which the statements in the slices are inspected. For Weiser’s slicing, we included all dependencies, whereas for thin slicing we omitted control dependencies and base-variable data dependencies (i.e., dependencies where the variable at the use is a field of a method of the object to which the method belongs) [36].

For thin slicing, we used breadth-first backtrafsals of the graphs to determine the order in which the statements in the slices are inspected. For Weiser’s slicing, we included all dependencies, whereas for thin slicing we omitted control dependencies and base-variable data dependencies (i.e., dependencies where the variable at the use is a field of a method of the object to which the method belongs) [36].

For these two techniques, to determine the ranking of a failing statement, we computed the minimum number of dependencies visited before reaching that statement during the traversal of the corresponding dependence graph (with the appropriate omissions for thin slicing). This number is the depth of the statement. For faults not found by thin slicing, we report the effort as not found (N/F).

5.3 **Results and Analysis**

Tables 5–8 present the effort of using the three variants of PriorSLICE, Weiser’s slice traversal (the WSlice column), and thin slicing (the Thin Slice column) to reach each faulty statement in NanoXML, XML-Security, JMeter, and Jaba, respectively. We also show four summary values:

<table>
<thead>
<tr>
<th>subject</th>
<th>fault ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>NanoXML</td>
<td>v1s1, v1s2, v1s3, v1s4, v1s5, v1s6, v1s7</td>
</tr>
<tr>
<td>XML-security</td>
<td>v1s2, v1s3, v1s5, v1s14, v1s16, v1s17, v1s20</td>
</tr>
<tr>
<td>JMeter</td>
<td>v2s1, v2s2, v2s5, v2s6, v2s7, v2s11, v2s19</td>
</tr>
<tr>
<td>Jaba</td>
<td>v1s2, v1s5, v1s7, v1s11, v1s12, v1s13, v1s17</td>
</tr>
</tbody>
</table>

http://www.nd.edu/~yzhang20/priosli
Table 5: Effort of using each technique to locate the fault in NanoXML

<table>
<thead>
<tr>
<th>faults in NanoXML</th>
<th>PRIOSLICE</th>
<th>WSlice</th>
<th>Thin Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1s1</td>
<td>27.2%</td>
<td>27.3%</td>
<td>24.4%</td>
</tr>
<tr>
<td>v1s2</td>
<td>31.8%</td>
<td>31.3%</td>
<td>29.6%</td>
</tr>
<tr>
<td>v1s3</td>
<td>33.8%</td>
<td>52.0%</td>
<td>34.0%</td>
</tr>
<tr>
<td>v1s4</td>
<td>27.1%</td>
<td>26.7%</td>
<td>18.0%</td>
</tr>
<tr>
<td>v1s5</td>
<td>15.8%</td>
<td>16.2%</td>
<td>6.7%</td>
</tr>
<tr>
<td>v1s6</td>
<td>21.6%</td>
<td>21.6%</td>
<td>9.8%</td>
</tr>
<tr>
<td>v1s7</td>
<td>16.6%</td>
<td>16.6%</td>
<td>9.4%</td>
</tr>
<tr>
<td>average</td>
<td>24.9%</td>
<td>27.4%</td>
<td>18.9%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>7.7%</td>
<td>12.2%</td>
<td>10.8%</td>
</tr>
<tr>
<td>average vs. thin slice</td>
<td>23.3%</td>
<td>22.3%</td>
<td>16.3%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>7.7%</td>
<td>7.5%</td>
<td>11.6%</td>
</tr>
</tbody>
</table>

Table 6: Effort of using each technique to locate the fault in XML-Security

<table>
<thead>
<tr>
<th>faults in XML-Security</th>
<th>PRIOSLICE</th>
<th>WSlice</th>
<th>Thin Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1s2</td>
<td>14.5%</td>
<td>15.4%</td>
<td>21.9%</td>
</tr>
<tr>
<td>v1s3</td>
<td>14.2%</td>
<td>8.7%</td>
<td>4.0%</td>
</tr>
<tr>
<td>v1s5</td>
<td>4.5%</td>
<td>7.9%</td>
<td>3.9%</td>
</tr>
<tr>
<td>v1s14</td>
<td>29.0%</td>
<td>25.7%</td>
<td>20.0%</td>
</tr>
<tr>
<td>v1s15</td>
<td>18.4%</td>
<td>11.3%</td>
<td>16.4%</td>
</tr>
<tr>
<td>v1s17</td>
<td>5.4%</td>
<td>8.0%</td>
<td>9.3%</td>
</tr>
<tr>
<td>average</td>
<td>14.7%</td>
<td>12.4%</td>
<td>13.7%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>8.3%</td>
<td>6.4%</td>
<td>9.3%</td>
</tr>
<tr>
<td>average vs. thin slice</td>
<td>10.4%</td>
<td>10.2%</td>
<td>11.7%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>6.3%</td>
<td>3.6%</td>
<td>7.5%</td>
</tr>
</tbody>
</table>

1. The average effort for all versions of each individual subject,  
2. The standard deviation of the effort for all versions of each individual subject,  
3. The average effort for the faulty versions of each subject for which we can compare thin slicing with PRIOSLICE, and  
4. The standard deviation of the effort for the versions for which we can compare thin slicing with PRIOSLICE.

An effort of 10%, for example, indicates that the technique can reach the faulty statement by traversing 10% of the program. For thin slicing, not found (N/F) indicates that this technique is not applicable for that fault because the fault can only be reached through at least one control dependence or base-variable data dependence.

We also tested the distribution and statistical significance of the results. First, we conducted Wilcoxon signed-rank one-tailed test [38] to assess with confidence whether our technique is more effective than static and thin slicing. We chose 0.05 as the significance level. Table 9 shows the statistical test results. In the last column, “success” indicates that the result is significant at p ≤ 0.05, where p is the p-value. Second, we created boxplot diagrams for each subject (Figure 5–8) to compare the distributions of effort for the PRIOSLICE variants and static slicing. The dividing line inside each box represents the median effort for each technique and the boxes themselves represent the second and third quartiles. The whiskers cover the other two quartiles, from the minimum to 25% and from 75% to the maximum value observed. For space reasons, we omit boxplot diagrams for the fault subsets applicable to thin slicing.

Regarding RQ2, Table 10 shows the average total running time and the corresponding standard deviation of each technique in seconds for all the faults in each subject.

5.3.1 RQ1: Effectiveness of PRIOSLICE

The results in Tables 5–8 indicate that the three variants of PRIOSLICE reach the faulty statement faster, on average, than the other techniques for the three largest subjects. For NanoXML, PRIOSLICE 3 outperformed static slicing, although PRIOSLICE 1 and PRIOSLICE 2 lagged behind by 0.4% and 1.9%, respectively. For the faults in which the results are comparable to thin slicing, the three variants of PRIOSLICE required less average effort to locate the fault than thin slicing.

Static slicing and thin slicing always exhibited larger variances than PRIOSLICE. One factor that explains these variances is the unpredictable dependence distance from the failure to the faulty statement, which affects breadth-first traversals more than smarter traversals such as those of PRIOSLICE. Naturally, however, it still takes PRIOSLICE a greater-than-average effort to reach the fault when that fault is far away from the failure. Another factor is the type of the dependencies between failure and fault. If these dependencies are mostly data dependencies, thin slicing requires a small effort (when it can find the fault). However, when control dependencies exist in some paths and play a role in propagating the error, Weiser’s traversal and PRIOSLICE have an edge.

We manually inspected specific cases in which Weiser’s approach was more effective than PRIOSLICE. For fault NanoXML-v1s1, for example, PRIOSLICE 3 traverses 24.4% of the program to locate the fault whereas Weiser’s slicing only needs 4.7%. The mini-

<table>
<thead>
<tr>
<th>faults in NanoXML</th>
<th>PRIOSLICE</th>
<th>WSlice</th>
<th>Thin Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2s1</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.04%</td>
</tr>
<tr>
<td>v2s2</td>
<td>9.4%</td>
<td>5.9%</td>
<td>14.1%</td>
</tr>
<tr>
<td>v2s5</td>
<td>0.1%</td>
<td>10.3%</td>
<td>23.7%</td>
</tr>
<tr>
<td>v2s6</td>
<td>0.04%</td>
<td>0.04%</td>
<td>0.04%</td>
</tr>
<tr>
<td>v2s7</td>
<td>19.6%</td>
<td>13.1%</td>
<td>22.2%</td>
</tr>
<tr>
<td>v2s11</td>
<td>17.8%</td>
<td>17.0%</td>
<td>23.2%</td>
</tr>
<tr>
<td>v2s19</td>
<td>17.7%</td>
<td>12.4%</td>
<td>20.3%</td>
</tr>
<tr>
<td>average</td>
<td>7.8%</td>
<td>7.8%</td>
<td>13.9%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>9.1%</td>
<td>7.1%</td>
<td>11.3%</td>
</tr>
<tr>
<td>average vs. thin slice</td>
<td>9.0%</td>
<td>7.8%</td>
<td>12.4%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>8.8%</td>
<td>8.9%</td>
<td>11.7%</td>
</tr>
</tbody>
</table>

Table 7: Effort of using each technique to locate the fault in JMeter

Table 8: Effort of using each technique to locate the fault in Jaba

<table>
<thead>
<tr>
<th>faults in JMeter</th>
<th>PRIOSLICE</th>
<th>WSlice</th>
<th>Thin Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2s1</td>
<td>28.9%</td>
<td>37.7%</td>
<td>40.8%</td>
</tr>
<tr>
<td>v2s5</td>
<td>6.6%</td>
<td>1.3%</td>
<td>11.6%</td>
</tr>
<tr>
<td>v2s7</td>
<td>1.5%</td>
<td>1.4%</td>
<td>2.8%</td>
</tr>
<tr>
<td>v2s11</td>
<td>18.5%</td>
<td>17.9%</td>
<td>39.7%</td>
</tr>
<tr>
<td>v2s12</td>
<td>36.3%</td>
<td>43.4%</td>
<td>31.3%</td>
</tr>
<tr>
<td>v2s13</td>
<td>6.3%</td>
<td>3.8%</td>
<td>17.9%</td>
</tr>
<tr>
<td>v2s17</td>
<td>39.9%</td>
<td>39.9%</td>
<td>51.8%</td>
</tr>
<tr>
<td>average</td>
<td>16.4%</td>
<td>17.6%</td>
<td>24.0%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>14.0%</td>
<td>18.9%</td>
<td>15.6%</td>
</tr>
<tr>
<td>average vs. thin slice</td>
<td>15.7%</td>
<td>16.6%</td>
<td>22.9%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>15.5%</td>
<td>19.3%</td>
<td>16.2%</td>
</tr>
</tbody>
</table>
maximum number of dependencies between failure and fault is four, and three of them are control dependencies. There are also many data-dependence dominated paths between other statements in the slice and the failing point. Therefore, in this particular case, and despite considering control dependencies, PriorSlice is lured away by more promising data dependencies while a simpler breadth-first search finds the fault faster.

With respect to thin slicing, PriorSlice was more effective for two reasons. First, PriorSlice guaranteed that all faults would be found, whereas thin slicing was not applicable in more than half of all cases. We confirmed that, in those cases that thin slicing could not locate the fault, the dependencies from the faulty code are control dependencies, which are discarded by thin slicing. Second, when thin slicing was applicable, PriorSlice was still better overall by discriminating among data dependencies to traverse slices faster towards the fault. However, in two cases, thin slicing gave greater priority than PriorSlice to apparently weak data dependencies which turned out to carry the error.

From the statistical test results in Table 9, we obtained confidence that the PriorSlice variants are more effective than the other techniques in 4 out of 6 cases. In the other two cases, the p-value is close to the significance level. Despite the smaller sample size (14 out of 28 faults), two PriorSlice variants were also significantly more effective than thin slicing. (Not to mention that PriorSlice is also applicable to the other 14 faults.)

From the distributions depicted in boxplots in Figures 5–8, we make two main observations. First, the median value of the effort of using static slicing is larger than using our techniques in most cases. The exceptions are the medians in the smaller subjects. The three variants of PriorSlice are more effective than the other techniques in 4 out of 6 cases. In the other two cases, the p-value is close to the significance level. Despite the smaller sample size (14 out of 28 faults), two PriorSlice variants were also significantly more effective than thin slicing. (Not to mention that PriorSlice is also applicable to the other 14 faults.)

With respect to thin slicing, PriorSlice was more effective for two reasons. First, PriorSlice guaranteed that all faults would be found, whereas thin slicing was not applicable in more than half of all cases. We confirmed that, in those cases that thin slicing could not locate the fault, the dependencies from the faulty code are control dependencies, which are discarded by thin slicing. Second, when thin slicing was applicable, PriorSlice was still better overall by discriminating among data dependencies to traverse slices faster towards the fault. However, in two cases, thin slicing gave greater priority than PriorSlice to apparently weak data dependencies which turned out to carry the error.

From the statistical test results in Table 9, we obtained confidence that the PriorSlice variants are more effective than the other techniques in 4 out of 6 cases. In the other two cases, the p-value is close to the significance level. Despite the smaller sample size (14 out of 28 faults), two PriorSlice variants were also significantly more effective than thin slicing. (Not to mention that PriorSlice is also applicable to the other 14 faults.)

From the distributions depicted in boxplots in Figures 5–8, we make two main observations. First, the median value of the effort of using static slicing is larger than using our techniques in most cases. The exceptions are the medians in the smaller sub-

ject, NanoXML, for two variants of PriorSlice, and PriorSlice 3 on Jaba, whose median is greater than the median of Weiser’s approach by about 5%. Second, Weiser’s approach has larger variation in all cases, as we also observed from the standard deviation results. From these boxplot distributions, we can confirm that the efforts of using PriorSlice to locate faults vary over a shorter range. Therefore, the graphs support our conclusion that the PriorSlice variants are not only a better choice overall but are also more stable.

5.3.2 RQ2: Practicality of PriorSlice

Table 10 presents the average total running time and the corresponding standard deviation of each technique for each subject. In all cases, our implementation of PriorSlice takes more time than Weiser’s slicing and thin slicing because it performs static slicing before moving on to the more complex process of estimating probabilities. For NanoXML, this difference was not too important—about 30 extra seconds on average for PriorSlice 3—but for other three larger subjects the difference was notorious. However, PriorSlice 1 is, on average, 5.95 times faster than PriorSlice 2 and 3.94 times faster than PriorSlice 3. For large subjects like JMeter and Jaba, the average running time of PriorSlice 1 still stays within an acceptable range (15 minutes). These numbers indicate a trade-off between the efficiency of the technique and the effectiveness of the technique due to the simplification in implementation. The three variants of PriorSlice provide the users options to choose in different scenarios. If a user, for example, wants to locate the bug in a nightly build software, she might have enough time to run the heavyweight but more effective technique, PriorSlice 2 or PriorSlice 3. However, if the user wants to get a quick result of
the prioritized slice, she can run the simplified version PRIOSLICE 1 with a still acceptable effectiveness compared to other existing techniques and an acceptable efficiency.

### 5.4 Threats to Validity
The main internal threat to the validity of the study is the possibility of implementation errors in PRIOSLICE or DUA-FORENSICS when slicing and computing the priorities for the subjects. However, this threat is mitigated by the maturity of DUA-FORENSICS, which has been developed for years and has been used in many other studies. We also tested PRIOSLICE carefully using examples and the study subjects because this is the newest part of our code base.

The main external threat to our study is the limited number of subjects and faults used. The 28 faults we used in our study, however, span a range of faults and ways in which those faults propagate to cause failures. Also, our subjects use different programming styles and have different purposes.

### 6. RELATED WORK

Program slicing was introduced as an analysis for program comprehension and debugging [20, 39]. Unfortunately, static slices are often too big to be useful. Our work alleviates this problem by recognizing that not all statements are equally relevant in a slice and that a static analysis can estimate their relevance to improve the effectiveness of the forward slice.

In preliminary work [31, 35], we described a forward version of probabilistic slicing which estimates the impacts of a change and creates a global ranking for inspecting impacted statements, regardless of their dependence distance from the slicing criterion. We have also investigated how to estimate, in particular, the occurrence of data dependencies [42]. PRIOSLICE, presented in this paper, is our next iteration in this line and the first to work backwards. Unlike previous work, PRIOSLICE uses probabilities to prioritize the traversal of dependence graphs, which mirrors what developers do [36, 39], and we showed that this new approach can outperform other static-slicing approaches.

A few other techniques discriminate among statements within slices. Two of them [16, 41] work on dynamic slices to estimate the influence of statements on outputs. These techniques, however, require runtime information which might not be readily available. Statically, thin slicing [36] distinguishes statements in slices pruning all control dependencies and pointer-based data dependencies (although pruned dependencies can be incorporated later). Our technique, in contrast, always keeps all statements from the static slice (which is a safe approach) and, instead, estimates their influence to prioritize the traversal of backward slices.

In addition to the techniques mentioned above, dynamic slicing [3, 24] and other variants of program slicing have been proposed. Conditioned slicing [11] lets users reduce slice sizes by restricting the input space, which is a feature PRIOSLICE currently does not provide. Researchers have also explored combining static slices with execution data, such as call stacks and dynamic aliasing [17, 19, 25, 27]. Although PRIOSLICE is designed on purpose to avoid execution data, lightweight forms of such data could refine PRIOSLICE.

Beyond slicing, there is a considerable amount of work on (semi-)automated fault localization. One related technique is the probabilistic dependence graph [6], which uses a statistical model of the runtime behavior of individual dependencies. Statistical approaches, based on code coverage, have been proposed and widely studied (e.g., [1, 12, 23, 33]). Spectrum-based fault localization [1], for example, uses an execution profile and uses a similarity coefficient to rank potential fault locations. Execution differencing [37] and delta debugging [40] are other well-studied approaches. Many of the approaches mentioned above require at least one test case to generate dynamic information, which makes these approaches essentially different from our technique. Our technique is purely static and requires only the program itself and the failing statement but no additional information.

### 7. CONCLUSION AND FUTURE WORK

In this paper, we presented PRIOSLICE, a new static-slicing technique that estimates the impact of each statement in a static backward slice to tell not only whether the statement is in the slice, but also how much it is that slice. These values for statements are used by PRIOSLICE to traverse the dependence graph by decreasing order of the estimated impacts of statements. We described the probabilistic model used to compute these values and explained our design decisions. We also develop three variants of PRIOSLICE to study the cost-benefits of design choices for PRIOSLICE. Our experimental results suggest that PRIOSLICE can, on average, spend less effort localizing faults than Weiser’s original slicing approach and thin slicing—even in the cases that thin slicing was applicable. We also conducted statistics tests which show that in 4 out of 6 cases, we have confidence that our technique is more effective than the others.

This work provides a new way of looking at slices, in which statements are distinguished by relevance or strength. This work can be used not only for fault localization, but also for program comprehension and speculative parallelization. Also, a forward version of this model can be used for change-impact analysis and test-case prioritization. Our probabilistic model has plenty of room for improvements on assigning weights with greater precision and, thus, greater effectiveness.

In the future, we plan to improve the precision of the model by using static branch-prediction heuristics (e.g., [7]) to better predict successor statements in our model. We also expect to extend our studies to more applications, such as program comprehension, to analyze the benefits for developers of our prioritized static slice.
8. REFERENCES


