Dynamic Path Conditions in Dependence Graphs

Christian Hammer [*]
University of Passau
Passau, Germany
hammer@fmi.uni-passau.de

Martin Grimme
University of Passau
Passau, Germany
grimme@fmi.uni-passau.de

Jens Krinke
FernUniversität in Hagen
Hagen, Germany
krinke@acm.org

Abstract

We present a new approach combining dynamic slicing with path conditions in dependence graphs enhanced by dynamic information collected in a program trace. While dynamic slicing can only reveal *that* certain dependences have been holding during program execution, the combination with dynamic path conditions reveals *why*, as well.

The approach described here has been implemented for full ANSI-C. It uses the static dependence graph to produce a fine-grained variable and dependence trace of an executing program. This information is used for dynamic slicing, yielding significantly smaller sets of statements than static slices, as well as for increasing precision of the path condition between two statements. Such a dynamic path condition contains explicit information about if and how one statement influenced the other.

Dynamic path conditions work even when tracing information is incomplete or corrupted e.g. in case of a "damaged flight recorder".

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging—Tracing; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Reliability, Security, Theory, Verification

Keywords Dynamic Slicing, Dynamic Chopping, Path Condition, Information Flow Control

1. Introduction

Security for a software product should always be guaranteed a priori to its deployment, at least for security-sensitive products. Traditionally, this task has been done by static program analysis techniques which provide powerful means to guarantee certain properties. For example, the ValSoft system [13] uses static program slicing to check if security relevant parts of the system are influenced by not security relevant parts and if such an influence has been found, ValSoft can generate necessary conditions for this influence to occur (called *path conditions*). Program slicing can be seen as a form of information flow control [16]. Still, such checks can only assert the validity of the specified properties.

For unforeseen incidents security-sensitive modules usually contain some sort of "flight recorder". It allows a posteriori reconstruction of problems leading to a—possibly fatal—error.

This work presents a new approach to employ the data recorded during program execution—the program *trace*—for a posteriori detection and isolation of problem causes. The trace is used to gain higher precision in two ways, which may as well be combined: First, a dynamic slicing algorithm identifies all statements that actually influenced the fatal statement during program execution. The dynamic slice is generally much smaller than the static slice and thus, a smaller set of statements have to be examined. If such a statement is suspicious, a path condition can be computed between the suspicious and the fatal statement. Path condition generation is based on a chop between the suspicious and the fatal statement. The dynamic chop between these statements is, again, generally much smaller than the static chop (chops contain the statements that participate in an influence from a source to a target statement). Thus, a dynamic chop contains a smaller number of paths between the two statements, leading to a less conservative path condition. Second, the observed values of program variables are transformed into an additional logical constraint, which, conjunctively combined, improves the precision of path conditions.

This dynamic path condition allows the precise reconstruction of the scenario that lead to the fatal error (*post-mortem analysis*). If the dynamic path condition is unsatisfiable, there was definitely no influence between the given statements even though the dynamic chop indicated otherwise. But if the path condition is satisfiable, it serves as a "witness" for the illegal information flow: A constraint solver will resolve the path condition to input values which triggered the illegal flow. These input values can be given to the program again and the influence becomes visible once more. In case of safety violations, these input values thus serve as witnesses for the illegal behavior.

The remainder of this paper is organized as follows. Section 2 presents the theoretical foundations of slicing and path conditions. In Section 3 we describe how information for the program trace is collected and discuss problematic points of tracing. Section 4 presents the variants of the dynamic slicing algorithm. Their application for path condition generation and the additional constraint based on dynamic variable data is described in Section 5. Experimental results are presented in Section 6. Related work is discussed in Section 7. The last section concludes and presents future work.

[*] This research was supported by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-1).

```
1 a = u();
2 while (n>0) {
3   x = v();
4   if (x>0)
5     b = a;
6   else
7     c = b;
8 }
9 z = c;
```

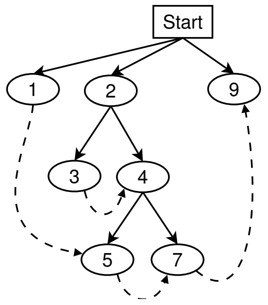

Figure 1. A small program and its dependence graph

2. Foundations

2.1 Static Slicing and Dependence Graphs

Mark Weiser introduced (static) program slicing primarily as a debugging aid [21]. The idea was that programmers mentally abstract away any code that cannot influence a statement showing unexpected behavior. He called this statement the *slicing criterion*. Weiser gave an algorithm for automatic slicing based on data flow analysis in the source code. Later, program slicing was defined as a reachability analysis in the *Program Dependence Graph (PDG)* [6].

Program dependence graphs are a standard tool to model information flow through a program. Program statements or conditions are represented by the nodes, the edges represent the dependences between statements or conditions. A *data dependence* edge $x \to y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A *control dependence* edge $x \to y$ means that the mere execution of y depends on the value of the condition x (which is typically a condition in an if- or while-statement).

A path $x \to^* y$ means that information can flow from x to y; if there is no path, it is guaranteed that there is no information flow. In particular, all statements (possibly) influencing y (the so-called *backward slice*) are easily computed as

$$BS(y) = \{x \mid x \to^* y\}$$

For the small C program and its dependence graph in Figure 1, there is a path from statement 1 to statement 9, indicating that input variable a will eventually influence output variable z. Since there is no path $1 \to^* 4$, there is definitely no influence from a to x.

A chop for a chopping criterion (x, y) is the set of nodes that are part of an influence of the (source) node x on the (target) node y. This is basically the set of nodes that lie on a path from x to y in the PDG:

$$CH(x, y) = \{z \mid x \to^* z \to^* y\}$$

For programs with procedures slicing and chopping is more complex because the calling context of procedures has to be obeyed. However, because the calling context is preserved in dynamic slicing and chopping almost automatically, it will not be discussed here.

Note that PDGs and slicing are much more complex for realistic languages with pointers, complex control flow, and data structures. An overview of fundamental slicing techniques can be found in [12, 18]; technical details will not be discussed here. For the full C language, the computation of precise dependence graphs and slices is absolutely nontrivial; there is ongoing research worldwide since 15 years. The state of the art in PDGs and slicing is summarized in the recent work by Krinke [11].

2.2 Path Conditions

In order to make the analysis more precise, Snelting et al. introduced *path conditions* [16], which are necessary conditions for information flow between two nodes.

The formulae for the generation of path conditions are quite complex (for details, see [16]), and only the most fundamental formula will be given here:

$$PC(x, y) = \bigvee_{P \text{ Path } x \to^* y} \quad \bigwedge_{z \text{ node in } P} E(z)$$

where $E(z)$ is a necessary condition for the execution of z:

$$E(x) = \bigvee_{P \text{ Control Path } Start \to^* x} \quad \bigwedge_{v \to \mu \in P} c(v \to \mu)$$

A control path is a path that consists of control dependence edges only. Thus, $E(x)$ is computed along all control paths from the *Start* node of the function to x based on the conditions $c(v \to \mu)$ associated with dependence edge $v \to \mu$. For control dependences, $c(v \to \mu)$ is typically a condition from a while- or if-statement. Program variables in a path condition are (implicitly) existentially quantified, as they are necessary conditions for potential information flow.

Because the paths between the criterion nodes are based on the computed chops, we assume that a chop $CH(x, y)$ is the set of paths between x and y. We will be interested in the set of paths $P_1, P_2, \ldots \in CH(x, y)$ and a slightly relaxed notation for path conditions is used:

$$PC(x, y) = \bigvee_{P \in CH(x,y)} \bigwedge_{z \in P} E(z)$$

Figure 1 shows a small example program fragment and its dependence graph. For this example, the following execution and path conditions are computed:

$$\begin{aligned}
&c(2 \to 3) \equiv c(2 \to 4) \equiv (n > 0),\\
&c(4 \to 5) \equiv (x > 0), \;\; c(4 \to 7) \equiv (x \leq 0),\\
&E(1) \equiv \mathit{true}, \quad E(3) \equiv (n > 0),\\
&E(5) \equiv (n > 0) \land (x > 0),\\
&PC(1, 5) \equiv E(1) \land E(5) \equiv \exists n, x.(n > 0) \land (x > 0)
\end{aligned}$$

In the presence of complex data structures like arrays or pointers, additional constraints will be generated. For data dependences, $c(v \to \mu)$ is a condition constraining information flow through data types. As an example we only consider arrays (a full presentation can be found in [16]): A data dependence $v \to \mu$ between an array element definition $a[E_1] = \ldots$ and a usage $\ldots = a[E_2]$ generates $c(v \to \mu) \equiv E_1 = E_2$; all other data dependences will generate $c(v \to \mu) \equiv \mathit{true}$. The equation to compute a path condition now becomes:

$$PC(x, y) = \bigvee_{P \in CH(x,y)} \bigwedge_{z \in P} E(z) \land \bigwedge_{v \to \mu \in P} c(v \to \mu)$$

For clarification consider the following program fragments and their path conditions:

```
1     a[i+3] = x;
2     if (i>10)
3         y = a[2*j-42];
```

$$PC(1, 3) \equiv \exists i, j.(i > 10) \land (i + 3 = 2j - 42)$$

and

```
1     a[i+3] = x;
2     if ((i>10)&&(j<5))
3       y = a[2*j-42];
```

$$\begin{aligned} PC(1,3) &\equiv \exists i, j.(i > 10) \land (j < 5) \\ &\qquad\qquad \land (i + 3 = 2j - 42) \\ &\equiv \mathit{false} \end{aligned}$$

These examples indicate that path conditions give precise conditions for information flow and can even determine that such flow is impossible even though there is a path in the graph.

Note that in practice path conditions tend to be large and a constraint solver is used to simplify them.

Details of path condition generation are not presented here, but the reader should be aware that making path conditions work for full C and realistic programs required years of theoretical and practical work [11, 14–16]. Just to mention a few things: the program must be transformed into single assignment form first (see below); and while PDG cycles can be ignored, due to the high number of cycle-free PDG paths in realistic programs, interval analysis for irreducible graphs must be exploited to obtain a hierarchy of nested sub-PDGs; BDDs must be used to minimize the size of path conditions. Today, our implementation ValSoft can handle C programs up to approx. 10000 LOC and generate path conditions in a few seconds or minutes.

2.2.1 Multiple Variable Assignments

Consider the example code in Figure 2 (left) and the (primitive) path condition

$$PC(1,5) \equiv (x < 7) \land (x = 8)$$

between `a` in line 1 and `x` in line 5. This condition is unsatisfiable, although there is definitely a way how line 1 can influence line 5. The problem is that the program contains multiple assignments to the variable `x` that this path condition cannot distinguish. For static path conditions this problem is solved by using a variant of SSA-form [5] of the program. That way, different variable definitions are distinguished and eventually brought together using the ϕ operator, thus replacing multiple variable assignments with single assignments. Figure 2 (right) shows the SSA form of the original program (left).

The SSA form makes our path condition solvable by distinguishing between different definitions of the variable x:

$$PC(1,5) \equiv (x_2 < 7) \land (x_3 = 8)$$

Transforming a program into SSA form, however, modifies the code representation and is thus not desirable for dependence graphs in ValSoft which are close to the source code structure. In order to maintain the code structure, an assignment form similar to the SSA form is used: Index numbers represent the node numbers in the dependence graph, allowing a precise distinction between different variable occurrences. Path conditions as

$$(e\_puf[idx] == \text{"}+\text{"})$$

are thus written as

$$(e\_puf_{99}[idx_{98}] ==_{97} \text{"}+\text{"}_{101})$$

The ϕ operator does not occur in the code structure itself, but is only used for computing path conditions.

2.2.2 Weak and Strong Path Conditions

For a given chop between two statements x, y one can usually define more than one path condition. Still, every single instance is a necessary condition for information flow along the chop. To argue about quality, a partial order[1] $\leq$ is defined for the pair (x, y)

$$PC'(x,y) \leq PC(x,y) \qquad \text{iff} \qquad PC(x,y) \Rightarrow PC'(x,y)$$

[1] In fact, path conditions form only a preorder. Modulo equivalence one obtains a partial order [14, 16].

```
1 x = a;                  1 x₁ = a;
2 while (x < 7) {         2 while (x₂=Φ(x₁,x₃),x₂<7){
3   x = y + x;            3   x₃ = y + x₂;
4   if (x == 8)           4   if (x₃ == 8)
5     p(x);               5     p(x₃);
6 }                       6 }
```

Figure 2. Multiple variable assignments

In such a case $PC(x,y)$ is called *stronger* than $PC'(x,y)$. Stronger path conditions are usually easier to solve by the constraint solver and thus more favorable.

Note that the precision of the underlying chop affects the strength of the path condition: if two chops exist where one is more precise than the other $CH(x,y) \subset CH'(x,y)$, then every path $P \in CH(x,y)$ in the smaller chop is also a path in the larger chop. Thus, the smaller chop generates a stronger path condition, since the disjunction in the path condition runs over fewer paths:

$$\bigvee_{P \in CH(x,y)} \bigwedge_{z \in P} E(z) \Rightarrow \bigvee_{P \in CH'(x,y)} \bigwedge_{z \in P} E(z)$$

This fact forms the theoretic basis for Section 4.

Adding another conjunctive term R to the path condition is a different way to strengthen it. In Section 5 logical formula will be generated from dynamic trace data and conjunctively combined with the original path condition, yielding a stronger (or equal) path condition:

$$PC(x,y) \land R \geq PC(x,y)$$

3. Program Tracing

Trace data, also known as a runtime protocol of variable bindings and their def-use locations, plays a role for dynamic slicing (cf. Section 4) and for refinement of path conditions (cf. Section 5).

To collect trace data one has to execute the program in a controlled environment, which motivated the employment of a standard debugger like the `gdb`. We implemented a debugger driver that abstracts away from the actual debugger in use, offering the tracer a standard interface for controlled execution.

The used tracing approach is based on a static dependence graph. Any information that the tracer (and the debugger) needs for controlled execution, like where to set break points and used/defined variables, are extracted from a fine-grained system dependence graph (SDG). Fine-grained means that statement nodes are expanded to an *Abstract Syntax Tree* [11]. This fine-grained structure forms a prerequisite for building path conditions in general. It also allows detailed tracing of variable bindings, where variables that need to be recorded before statement execution (variables used for the computation) are distinguished from the variable(s) defined by the statement, which is recorded after execution. Thus every statement is mapped to a set of variables and their role (Definition, Use). The control dependence information is extracted from the SDG.

In the tracing phase, the program is executed statement by statement, where for every statement the attached variables are traced, either before or after the execution of the statement. For procedure calls the tracer maps the actual parameters to the formal parameters. This implies a Use and Definition role at the same time, which are traced before the execution of the method call. Note that a trace 'inlines' the called procedures and thus, is automatically context-sensitive.

```
1  LINE 11               1  LINE 11
2  USE Z                 2  USE Z
3  DEF X                 3  DEF X
4  LINE 12               4  LINE 12
5  USE X                 5  USE X
6  LINE 13               6  LINE 13
7  USE Y                 7  USE Y
8  DEF X                 8
9  LINE 14               9  LINE 14
10 USE X                 10 USE X
```

Figure 3. Incorrect dependence by gap in protocol

3.1 Third Party Code

A problem well known in static program analysis arises for dynamic analyses as well: Libraries (especially provided by a third party) usually do not provide source code nor the debugging information needed to collect tracing data. I.e. any side-effect produced by a library call does not generate the tracing information to produce correct dynamic dependences. When the debugging information is extracted from the static SDG that problem arises already during construction of the SDG. But even if one did not depend on a static dependence graph would one face the same problem.

A possible solution has been employed by static analysis designers for some years now: One writes stubs for those library methods and conservatively adds the summary dependences at the invocation point.

3.2 Incomplete Traces

Besides the problem of third party code, other reasons exist, why a trace could be incomplete: Either the tracer looses information, maybe on purpose for restricted memory, or because of limitations of the tracing approach. But it depends on the purpose whether the detail of the traced data suffices to gain sound results. As mentioned at the beginning of this section, our goals are dynamic slicing resp. chopping (cf. Section 4) and the refinement of path conditions with dynamic variable data (cf. Section 5).

Dynamic slicing does not depend on the actual values of variables but on the def-use relations of variables. Missing entries in the trace will most probably lead to false dependences and thus incorrect dynamic slices. As an example consider Figure 3. While in the left protocol line 10 depends on the definition in line 8, this entry has been missed in the protocol on the right. The dynamic slice will determine a dependence to line 3 then, which is incorrect.

With `gdb` controlling the program execution and the fine-grained variable tracing, one cannot guarantee the correctness of the program trace in all cases: First one has to assert that one line of source code has not more than one statement. Code like `x = a + x; x++;` will not result in a detailed protocol since the debugger works only line-based and will thus report only one definition and one use of `x` instead of two, respectively. Tools like GNU indent produce code that circumvents these problems.

Multiple assignments to the same variable in one statement like `x = a + x++` are undefined in ANSI-C and will thus be ignored. Under certain circumstances, however, our technique will produce fragmentary traces in special cases: a statement with two method calls like `x = f(a) + g(b)` may yield an incorrect value for `b` as the debugger cannot stop between the method calls to allow accurate parameter tracing. A solution to this problem is the combination of static slicing and dynamic slicing similar to [23]. Another solution would be to transform the source program to a program that has at most one assignment or one function call per statement.

4. Dynamic Slicing

Dynamic program slicing was introduced by Korel and Laski [10]. Dynamic slicing builds a *dynamic dependence graph* computed from the real dependences arising during program execution. Therefore, a dynamic dependence graph usually is considerably smaller than a static dependence graph, which has to relay on conservative approximations not to relinquish soundness.

For illustration consider Figure 1 again. If the execution trace is 1,2,3,4,7,9 then the static backward slice of node 9 is the whole graph. The dynamic slice of 9 does, in contrast, not contain the statements 1 and 5 as those did not contribute to the value of `z` in the given run.

Once a program trace has been collected, dynamic slicing typically falls to two tasks: In the *preprocessing* phase a dynamic dependence graph is generated by processing the collected data. In the *slicing* phase this graph is traversed to build the dynamic slice for the given slicing criteria.

A naive approach to dynamic slicing would mark all statements encountered during program execution, reduce the static dependence graph to the corresponding nodes, and do static slicing on that graph. This approach is, however, imprecise which can be illustrated on Figure 1: With the execution trace 1,2,3,4,7,2,3,4,5,9 the naive algorithm would mark all nodes visited, yielding a dynamic slice that contains the whole graph. Node 5 had no effect on node 7, though, as the definition of `b` took place after the use. So, nodes 1 and 5 should not be in the dynamic slice.

As a remedy, Agrawal [1] proposed not to work on the static dependence graph but on the tracing protocol, which shows a linear program with all loops unrolled. From that data the dynamic dependence graph needs to be computed. Its nodes usually represent basic blocks rather than single statements, which build the nodes of the static variant. Dependences point from a variable use to its last definition. It may be a bit confusing that dynamic edges are reversed compared to static edges. Dynamic (backward) slicing thus follows all edges starting from the slicing criterion:

$$dBS(y) = \{x \mid y \rightarrow^* x\}$$

Since the length of the runtime protocol is in principle unbounded, the space requirement of the context-sensitive dynamic dependence graph for long program runs explodes. Therefore several ways to compact this graph were proposed.

Agrawal [1] noted that the number of statements in a program is bounded and hence, the number of different slices must be bounded, too. He found that nodes with the same transitive dynamic dependences could be merged. This graph was called *Reduced Dynamic Dependence Graph (RDDG)*. While this representation is quite compact and gives a program slice in $O(1)$ (the transitive dependences are stored in every node), different instances of the same node (e.g. in a loop) cannot be distinguished. So the reduced size of the graph results in a loss of precision.

Context-sensitivity is a property that is not granted with such an approach. As a consequence of the linearity of the trace, however, dynamic slicing can be done in a context-sensitive manner, if labels are added to the edges [24, 25]. The labels contain additional information to disambiguate the distinct execution instances of the statements that the edge links. Zhang et al. call this dynamic graph the *dynamic data dependence graph*. As an example consider Figure 2 together with the execution trace 1,2,3,4,2,3,4,5. The graph contains edge labels that capture the execution time of the involved statements. The check whether `x<7` on line 2 is executed at time 2 and depends on the value of `x` computed in line 1 at time 1. Thus the edge contains these timestamps: (2, 1). The node corresponding to line 3 (we will use the terms node/statement/line interchangeably for this example) is dependent on the execution of statement 1 in the first instance of the while-loop, represented by an edge marked

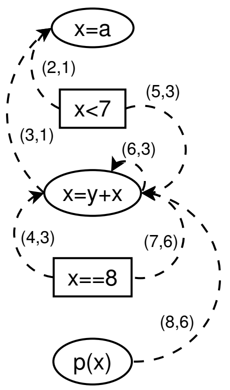

Figure 4. Dynamic data dependence graph for figure 2. Control dependence edges are omitted for readability

(3, 1); in the second it is dependent on the last instance of itself: the loop edge is marked with the execution times (6, 3).

The dynamic program slice is computed in the dynamic data dependence graph using the following formula (let $x \rightarrow_l y$ denote the edge from x to y with the timestamp label $l = (t_1, t_2)$):

$$\begin{aligned} dBS_{cs}(y,t) \quad = \quad & \{x \mid \exists \text{ Path } p : (y = x_0 \rightarrow_{l_0} \ldots \rightarrow_{l_{n-1}} x_n = x) : \\ & l_0 = (t, t_0) \ \wedge \\ & \forall 0 < i < n-1 : l_i = (t_{i-1}, t_i) \wedge l_{i+1} = (t_i, t_{i+1})\} \end{aligned}$$

In our example, the dynamic slice of the first execution of line 2 (with timestamp 2) is line 2 itself and line 1. Line 3 is not included, as the edge with timestamp 5 is not followed. Starting from line 2 with timestamp 5, however, we will have to include line 3 and come back to line 1. This small example already illustrates the power of edge labels.

Similar to the dynamic program slice, it is possible to define a dynamic program chop in the dynamic data dependence graph:

$$\begin{aligned} dCH_{cs}(x,y,t_x,t_y) \quad = \quad & \{x_i \mid \exists \text{ Path } p : (y = x_0 \rightarrow_{l_0} \ldots \rightarrow_{l_{n-1}} x_n = x) : \\ & l_0 = (t_y, t_0) \wedge l_{n-1} = (t_{n-2}, t_x) \ \wedge \\ & \forall 0 < i < n-1 : l_i = (t_{i-1}, t_i) \wedge l_{i+1} = (t_i, t_{i+1})\} \end{aligned}$$

A dynamic chop contains all nodes that are part of a path from x to y in the dynamic data dependence graph that starts at y with timestamp t_y and ends at x with timestamp t_x.

The dynamic data dependence graph is not restricted in space, though, and the graph can only be built if the runtime protocol is entirely processed which may take too much time for long-running applications. Zhang et al. [24, 25] thus proposed—apart from this *full preprocessing* algorithm (FP)—two variants that do not build the graph beforehand: *no preprocessing* (NP) and *limited preprocessing* (LP). The NP algorithm entirely forbears from constructing the dependence graph and, when slicing, runs back the linear trace to find the most recent definition of the given variable. Hence, NP has a worst case complexity of $O(N^2)$ which is unacceptable for large slices or for many slicing criteria. Even with the use of caching by marking statements already in the slice not to be followed again, this complexity cannot be lowered. The best compromise between the FP and NP algorithms is, according to the authors, the LP algorithm, which introduces summary information at a given offset between two such entries in the tracing protocol. Still, the complexity cannot be reduced by an order of magnitude but only by a constant factor.

We implemented all the mentioned algorithms for dynamic slicing and evaluated them on our test suite (cf. Section 6). Our experiments approve the results of Zhang et al. [24,25]. On average over 100 slicing criteria for agrep, the cached LP algorithm was, with about 20 seconds and 4.7 MB memory, faster than NP with about 21 sec and FP with 26 sec, consuming insignificantly more memory than the NP algorithm, which used 4.4 MB ram, and better than FP needing 6.4 MB.

5. Dynamic Path Conditions

5.1 Refinement by Dynamic Chopping

When constructing a path condition from a statement to another, all paths between those two statements are determined with a chop in the static dependence graph. As mentioned in Section 2.2.2, the accuracy of the path condition for the executed program can be increased if a dynamic chop is used instead. The dynamic dependence graph usually contains only the dependence edges that actually took place and thus the dynamic chop will yield a much smaller number of paths between those two statements.

As an example, consider Figure 1 again. If the execution trace is 1,2,3,4,7,9 then the dynamic chop between 2 and 9 is 2,3,4,7,9. Statement 5 is never executed in this setting and thus can be removed from the dynamic chop. A static chop would conservatively have to add it. With the dynamic chop one omits the paths 2,3,4,5,7,9 and 2,4,5,7,9 in the path condition between 2 and 9.

Although our path condition generator reuses partial information and thus half the number of paths does not yield a 50% shorter path condition, this small example already shows the impact of this refinement.

5.2 Refinement by Traced Values

In order to strengthen a given path condition PC with runtime information, the trace is analyzed to retrieve the variable assignments. As the analyzed program went through a series of assignment states during runtime, all of them have to be captured in a restrictive clause R. This clause, in turn, can be used to make path conditions stronger.

At first the intersection V of variables used in the path condition PC and the trace T is determined:

$$V = \{v \mid v \in var(T) \cap var(PC)\}$$

For each variable v the values it carried during the trace are extracted, let $\beta(v)$ the set of values w_i that variable v has contained:

$$\beta(v) = (v = w_1) \vee (v = w_2) \vee \ldots$$

Now the restrictive clause R can be described as the conjunction of all variable value sets:

$$R = \bigwedge_{v_i \in V} \beta(v_i)$$

In order to make the path condition PC stronger, the results from Section 2.2.2 are used and both clauses are conjunctively combined

$$PC' = PC \wedge R$$

yielding the stronger and thus more precise path condition PC'.

Because of the SSA-like form, variables only match if their index numbers are the same, so that multiple assignments are handled. This makes it mandatory for the program trace to list variables along with their respective node numbers as in Figure 5 which shows a simple trace for a run of the program from Figure 2.

Note that the node numbers are different to the SSA-numbers from Figure 2 and thus, the path condition for a flow from line 1 to line 5 is $PC(1,5) = (x_3 < 7) \wedge (x_7 = 8)$. For this path condition the trace yields the variable assignments

$$R = (x_3 = 2 \vee x_3 = 4 \vee x_3 = 6 \vee x_3 = 8) \wedge (x_7 = 4 \vee x_7 = 6 \vee x_7 = 8)$$

```
LINE 1
USE a_2 2
DEF x_1 2
LINE 2
USE x_3 2
LINE 3
USE y_5 2
USE x_6 2
DEF x_4 4
LINE 4
USE x_7 4
LINE 2
USE x_3 4
LINE 3
USE y_5 2
USE x_6 4
DEF x_4 6
LINE 4
USE x_7 6
LINE 2
USE x_3 6
LINE 3
USE y_5 2
USE x_6 6
DEF x_4 8
LINE 4
USE x_7 8
LINE 5
USE x_8 8
LINE 2
USE x_3 8
```

Figure 5. A simple program trace for Figure 2

Again, both clauses are conjunctively combined to $PC' = PC \wedge R$ yielding the stronger and thus more precise path condition PC':

$$PC' = (x_3 = 2 \vee x_3 = 4 \vee x_3 = 6) \wedge (x_7 = 8)$$

5.3 Correctness of Dynamic Path Conditions

Sometimes only fragments of a trace are available due to a "defective recorder" or intentionally to save memory. While fragmented traces are generally useless for dynamic slicing (see Section 3.2), they still hold valuable information for strengthening path conditions.

However, incomplete tracing information is prone to lead to wrong path conditions. For example, consider the simple path condition $(x > 1)$ for a program where the trace yields the restrictive clause

$$(x = 0 \vee x = 1)$$

while the variable assignment states actually were

$$(x = 0 \vee x = 1 \vee x = 2)$$

The restricted path condition

$$PC' = (x > 1) \wedge (x = 0 \vee x = 1) \equiv false$$

would be in contradiction to the actual program state $(x = 2)$ and thus definitely rules out data dependence where it may actually be possible.

To avoid unsound path conditions, it is conservatively assumed that there is an additional *unknown* value $\perp$ for each variable representing the assignments which occurred but were not traced due to some reason. This measure yields a correct conservatively restricted path condition being as precise as the fragmentation of the trace allows. For our example, the resulting path condition is

$$(x > 1) \wedge (x = 0 \vee x = 1 \vee x = \perp)$$
$$\equiv (x > 1 \wedge x = \perp) \equiv x > 1$$

Only if the completeness of the trace (at least for certain variables, see Section 3.2) can be guaranteed, one may abandon this conservative measure (for those variables).

It may seem that using this trick one doesn't gain any additional information of dynamic variable data. To show the advantage of variable traces containing unknown values, consider the path condition PC(1,5) of Figure 1. With a fragmented variable trace forming the conservative restrictive clause $(x = 5 \vee x = \perp) \wedge (n = 3 \vee n = \perp)$ the improved path condition from 1 to 5 will be:

$$PC(1,5) \equiv (n > 0 \wedge x > 0) \wedge (x = 5 \vee x = \perp) \wedge (n = 3 \vee n = \perp)$$

It is immediately clear that the traced variable values $x = 5$ and $n = 3$ may trigger an influence from line 1 to line 5.

This tiny example shows that while conservative restrictive clauses cannot be used to evaluate a clause of the path condition to *false*, they may reveal input values that triggered an illegal information flow.

6. Case Studies

Five case studies will show the impact of dynamic information on path conditions for actual programs. Table 1 lists the programs used for evaluation purposes together with lines of code and the number of nodes and edges in the (static) SDG. The programs `ptb_like` and `mergesort` are included in this article (figures 6 and 12). The remaining programs are taken from the GNU project.

Program	LOC	Nodes in SDG	Edges in SDG
ptb_like	35	134	334
mergesort	59	244	640
cal	678	2388	6149
agrep	3990	22961	81203
patch	7998	30774	246754

Table 1. Example programs for case studies

Program	static		dynamic		criterion
ptb_like	65	173	49	124	9-8, 33-53
mergesort	123	299	97	216	45-14, 21-8
cal	240	648	0	0	228-10, 281-18
	134	315	44	92	367-12, 551-3
agrep	13170	40324	0	0	605-15, 638-7
(sgrep.c)	13138	40144	961	2345	96-14, 121-9
patch	16529	246754	6314	81365	825-23, 935-10

Table 2. Evaluation of static vs. dynamic chop sizes

Our first goal was to show the impact of dynamic chopping in contrast to static chopping. Remember from sections 2.2.2 and 5.1 that smaller chop sizes result in more precise path conditions. Table 2 shows the number of nodes and edges for the static chop followed by these numbers for the dynamic chop. The chopping criterion is given in the format *from: line-column, to: line-column*. For the program `agrep` the criteria refer to the file `sgrep.c`. They were chosen in a way to find statements in the code which involve several variables that possibly influence each other, preferably in loops. The goal was to produce interesting path conditions. For example, the static chop in the program `ptb_like` (listed in Figure 6)

```
1  #define TRUE 1
2  #define CTRL2 0
3  #define PB 0
4  #define PA 1
5  void printf();
6  void main()
7  {
8    int p_ab[2] = {0, 1};
9    int p_cd[2] = {1, 1};
10   char e_puf[8] =
       {'0','0','0','0','0','0','0','0'};
11   int u = 0;
12   int idx = 0;
13   float u_kg = 0.0;
14   float kal_kg = 1.0;
15
16   while(TRUE) {
17     if ((p_ab[CTRL2] & 0x10)==0) {
18       u = ((p_ab[PB] & 0x0f) << 8) +
           (unsigned int)p_ab[PA];
19       u_kg = (float) u * kal_kg;
20     }
21     if ((p_cd[CTRL2] & 0x01) != 0) {
22       for (idx=0;idx<7;idx++) {
23         e_puf[idx] = (char)p_cd[PA];
24         if ((p_cd[CTRL2] & 0x10) != 0) {
25           if (e_puf[idx] == '+')
26             kal_kg *= 1.01; /* illegal */
27           else if (e_puf[idx] == '-')
28             kal_kg *= 0.99; /* illegal */
29         }
30       }
31       e_puf[idx] = '\0';
32     }
33     printf("Article:␣%7.7s\n" +
         "␣␣␣%6.2f␣kg␣␣␣␣",e_puf,u_kg);
34   }
35 }
```

Figure 6. ptb_like

```
( ((p_cd[0] & 0x01) != 0)
∧ ((p_cd[0] & 0x10) != 0)
∧ ((p_ab[0] & 0x10) == 0)
∧ (e_puf[idx] == '+')
∧ (idx < 7) )
∨
( ((p_cd[0] & 0x01) != 0)
∧ ((p_cd[0] & 0x10) != 0)
∧ (e_puf[idx] == '-')
∧ ((p_ab[0] & 0x10) == 0)
∧ (e_puf[idx] != '+')
∧ (idx < 7) )
```

Figure 7. Static path condition for ptb_like

```
... ∨
( e_puf == <unknown>
∧ idx == 0
∧ p_cd == {1, 1}
∧ p_ab == {0, 1}
∧ ((p_cd[0] & 0x01) != 0)
∧ ((p_cd[0] & 0x10) != 0)
∧ (e_puf[idx] == '-')
∧ ((p_ab[0] & 0x10) == 0)
∧ (e_puf[idx] != '+')
∧ (idx < 7) )
∨ ...
```

Figure 8. Excerpt of a dynamic path condition for ptb_like

```
  ((p_cd[0] & 0x01) != 0)
∧ ((p_cd[0] & 0x10) != 0)
∧ ((p_ab[0] & 0x10) == 0)
∧ (e_puf[idx] == '+')
∧ (idx < 7)
```

Figure 9. Dynamic path condition of illegal flow in ptb_like

from line 9 to line 33 (u_kg) contains 65 nodes connected by 173 edges. The dynamic chop, however, contains only 49 nodes and 124 edges. It is clear that the latter subgraph contains a noticeable smaller number of paths than the subgraph induced by the static chop. Sometimes the dynamic chop can rule out a dependence between two statements completely: Consider the first lines of the programs cal and agrep. One can see that the dynamic chop for these criteria is empty. The following evaluation of using traced variable values to improve precision in dynamic path conditions contains another example of that kind. In all these cases there was definitely no (illegal) information flow between the chopping criteria although the static chop indicated so.

After showing that dynamic chopping can considerably reduce the number of paths and thus yield a more precise path condition, that narrows down the reasons for an (illicit) influence, we will present excerpts of path conditions and augment them with the restriction condition, which is based on trace data (partly using incomplete traces). Again, this information reveals variable values which may have contributed to the necessary condition that triggered an illegal information flow.

The first example ptb_like, shown in Figure 6, is taken from a weighing machine controller. Such a program represents perfectly the security relevant software we have in mind for this approach: There is a part of the program, the so-called calibration path, that contains all paths from the sensor (p_ab) to the value display, in this case the weight stored in u_kg (line 33). For a certificate that the machine is correctly calibrated one needs to assure that there is no way to influence the calculation of the weight, for example from the keyboard. Consider the static path condition between the keyboard buffer p_cd in line 9 and the display of u_kg, the actual weight, in line 33. The final path condition is shown in Figure 7. Note that & represents bitwise logical and. An illegal information flow could only happen, if the keyboard buffer p_cd contained one of the special characters '+' or '-'. Since our run uses an input buffer containing only ones, we expect the dynamic path condition to evaluate to *false*.

Using a dynamic chop immediately reveals this fact: the static path condition describes a path that was not taken during runtime, the precise *dynamic* path condition yields *false*. The dynamic path condition based on the *static* chop, in contrast, consists of ten conjunctive blocks, one of which is shown in Figure 8. Dynamic trace data is shown in bold, e_puf has not been traced (incomplete trace). As one can see, the particular predicates

$$p\_cd == \{1, 1\}$$

```
  (left < right)
∧ (idx1 <= ((left + right) / 2))
∧ (data[idx1] >= data[idx2])
∧ (idx2 <= right)
```

Figure 10. Static path condition for mergesort

and

$$((p\_cd[0] \ \& \ 0x10) \neq 0)$$

contradict each other, so that the given block evaluates to *false*. The same goes with the other blocks and we get the expected result *false*. The path condition cannot be fulfilled; the necessary path was not taken. This result proves that the keyboard buffer had no influence on the output presented to the consumer.

In another scenario, the input `p_cd = {0xff, '+'}` (instead of line 9) has been traced. Upon entering '+' on the keyboard, the displayed value is too high. With the dynamic path condition the detection of the illicit influence is done automatically: Figure 9 shows the path condition for the adapted program based on the dynamic rather than the static chop. With the traced input one can exactly determine why the illicit information flow took place. Together with the definition of `e_puf[idx]` in line 23, adding the restrictive condition yields:

```
  (p_cd[0] == 0xff)
∧ (p_ab[0] == 0)
∧ (p_cd[1] == '+')
∧ (idx < 7)
```

This path condition already shows why there was an illegal information flow during program execution giving detailed information why the program produced incorrect output (the weight on the machine): The display was influenced by some debug flags and the input of '+' during the 7 rounds of the for-loop. The programmers simply had forgotten to remove the debugging code from the final version. This information can act as a witness to reproduce the illicit behavior. In this small example one can easily see that the calibration factor `u_kg` is increased in line 26 by such an input. For larger examples a human would most probably not detect illegal statements so easily.

As another example, consider the program `mergesort` from Figure 12. Figure 10 shows the statically computed path condition between `999` in line 45 and `temp` in line 21. This time, using a dynamic chop does not help strengthening the path condition as the dynamic chop is identical to the static chop regarding the paths relevant to the path condition. The dynamic path condition, however, yields 40 conjunctive blocks, one of which is shown in Figure 11. Dynamic trace data is again shown in bold. Due to contradictions within the particular blocks, the condition can be fully evaluated to *false*: there was no program state traced which would have fulfilled the static path condition.

As our examples show, dynamic path conditions are usually a good deal bigger than their statically computed counterparts, but also more precise as they hold more information. Each dynamic path condition is tied to a particular program run, though. If a dynamic path condition evaluates to *false* this does not necessarily hold for each program execution, especially if user input is involved. But dynamic path conditions are a precise means for finding witnesses for illegal program behavior.

Omission errors

Dynamic slices can only show *that* some influence took place or not. Sometimes one would like to know *why* an expected influence did *not* happen during program execution. In literature [2, 7, 20] several approaches were proposed to enrich the dynamic slice with the "culpable" control predicates, i.e. those predicates that triggered a branch to an execution path on which the expected potential data dependence did not come into effect. But adding only the predicates to the slice does not reveal what actually went wrong. Our approach returns the exact conditions for a data flow to happen. The static path condition augmented with the restrictive clause can be fed into a constraint solver to detect the contradictions between the variable trace and the values expected by the predicates. In Figure 11, the subclauses 1 and 2 contravene the sub-clause 7. That is the reason why the path(s) yielding that conjunctive block have not been executed. As pointed out in the description of `mergesort`, the dynamic chop is identical to the static chop. Adding only the control predicates as proposed by previous solutions would just reveal the conditions shown in Figure 10. The user would have to find those conditions in the slice and interpret them to find the information he or she was really looking for: the condition in Figure 11. The dynamic path conditions thus helps localizing flaws in the program by detecting contradictions between expected and actual execution paths.

```
  right == 4
∧ left == 4
∧ data == 1
∧ idx2 == 5
∧ ((left + right) / 2) == 3
∧ idx1 == 3
∧ (left < right)
∧ (idx1 <= ((left + right) / 2))
∧ (data[idx1] >= data[idx2])
∧ (idx2 <= right)
```

Figure 11. Excerpt of a dynamic path condition for mergesort

7. Related Work

Correctly and efficiently collecting trace data is a non-trivial task. Several solutions have been proposed in literature:

Venkatesh [19] implemented a low-level approach for tracing C programs. His prototype implementation called SLICE instruments the source code to write a program trace during execution. With this experience, the authors recommend object-code instrumentation for future implementations together with several reasons. Nonetheless, this implementation is faster than tracing with a traditional debugger.

Zhang et al. [24] follow a different low-level approach to create a program trace: The program source is compiled with the Trimaran system, a compiler for the Explicitly Parallel Instruction Computing (EPIC). An interpreter takes the generated object code and creates the program trace during execution. Since C programs are normally not interpreted, this approach is valid mostly for theoretical evaluations.

All low-level approaches usually do not slow down program execution as much as a debugger does. However, as our work is based on the static dependence graph which must be mappable to source, these approaches did not suit our needs.

Dynamic program slicing has been a topic in active research for several years now. Various approaches, either for dynamic slicing on its own, or combined with static elements have been proposed. To mention all of them would be out of scope of this work.

Chen et al. [3] describe a dynamic slicing algorithm that is based on a static PDG providing the information where to set break points for the debugger. The static dependence graph is confined

```
int data[100];
int temp[100];

void move (int *from, int fst, int lst,
           int *to, int idx) {
  while (fst <= lst)
    to[idx++] = from[fst++];
}

void merge (int fst, int mid, int lst) {
  int idx, idx1, idx2;

  idx = 0;
  idx1 = fst;
  idx2 = mid + 1;

  while ((idx1 <= mid) && (idx2 <= lst)) {
    if (data[idx1] < data[idx2])
      temp[idx++] = data[idx1++];
    else
      temp[idx++] = data[idx2++];
  }

  if (idx1 > mid)
    move (data, idx2, lst, temp, idx);
  else
    move (data, idx1, mid, temp, idx);

  move (temp, 0, lst - fst, data, fst);
}

void mergesort (int left, int right) {
  int m;
  m = (left + right) / 2;
  if (left < right) {
    mergesort (left, m);
    mergesort (m + 1, right);
    merge (left, m, right);
  }
}

int main () {
  int i;

  data[0] = 999;
  data[1] = 1;
  data[2] = 23;
  data[3] = 55;
  data[4] = 44;

  mergesort (0, 4);

  for (i = 0; i < 5; ++i) {
    printf ("%d␣", data[i]);
  }
  printf ("\n");

  return 0;
}
```

Figure 12. mergesort

to the nodes and edges that have been visited to build the dynamic dependence graph. Slicing is done following all edges in that graph.

Tip [17] embarks on a different strategy: He uses the abstract syntax tree (AST) instead of a dependence graph and interprets the program. The approach is language independent but only available for the custom-built language "L".

Zhang et al. recently proposed another way to reduce the vast amount of data that is stored in the program trace. They compute the dynamic slices during program execution and store them in binary decision diagrams (BDDs) [23].

Wang et al. [20] presented a dynamic slicing technique for Java that compresses the program trace on-the-fly and obtains two to three orders of magnitude compression with little overhead. A lossless compression algorithm finds a high repetition pattern in the sequence of (memory and control) addresses captured by the tracer separately. They also propose a dynamic slicing algorithm which operates directly on the compressed data and can thus save the uncompressing time. Such an algorithm may not only be suitable for languages with extensive pointer usage. We expect the repetition pattern for our variable trace to yield a similar compression rate. Since the slicing algorithm runs on the compressed data with no dynamic dependence graph used, multiple slicing requests require traversing the trace multiple times at a significant time overhead.

Recent work by Jhala has been focusing on path slicing [9]. It takes as input one particular path in the CFG and eliminates all the operations that are irrelevant towards the reachability of the target location. The result is a sufficient condition for the reachability of the target location, its infeasibility is sufficient for the infeasibility of the path. The technique does not work on the PDG but on the CFG only. It has shown effective for elimination of counterexamples provided by the model checker Blast. For our application this approach does not seem beneficial as it needs to check every single path on its own, while path conditions produce a necessary condition for all paths between two statements.

8. Summary and Future Work

This ongoing work has shown that dynamic tracing data can yield detailed reasons for or against a possible information flow from one statement to another. Especially in recently discovered applications of dynamic slicing, like improving software quality and security, such a condition may be able to rule out an undesirable or even illegal influence between two statements. If the condition cannot be evaluated to false, our approach yields precise information, why the information flow took place: A constraint solver will reduce the path condition to input values that triggered the illegal information flow. Those values form a witness for reconstruction of illegal influences.

Our approach presented two possibilities to refine a static path condition: First, dynamic slicing (more precisely chopping) greatly reduces the number of control paths. As these paths form the basis of the path condition computation, the number of its subclauses diminish. Second, the variable trace is transformed to a logical formula, the restrictive clause, which is conjunctively linked to the path condition. Combining these methods yields the greatest effect.

Nonetheless, the second approach is applicable even in the case that the variable trace is fragmentary. Whereas a dynamic slice would most probably yield incorrect dependences, the restrictive clause formed of a partial variable trace is still conservative. This corresponds to the principle of the flight recorder, which, even if defect, reveals valuable information to determine the problem cause.

The research on dynamic program slicing has not stopped. To make dynamic slicing applicable for more realistic programs, the size of the dynamic dependence graph can be reduced drastically. The recent work of Zhang and Gupta [23] shows a technique to to so: Their approach basically augments the dynamic dependence graph with pre-computed dependence edges determined by static analysis. These static dependence edges do not have to wear labels, hence all dynamic edges can be folded to a single static edge. We plan to improve our algorithms in such a way, especially as our fine-grained dependence trace contains lots of local dependences for edge sharing and the trace uses the static dependence graph already as input. Problems arising from third party library code and statements containing multiple method calls can be remedied that way as well.

Furthermore, we want to extend this work to the Java slicer [8], which recently has been integrated into ValSoft. As a necessary step towards that task we will augment the path condition generator to Java peculiarities. Dynamic slicers for Java like [20] are just becoming mature and can be integrated into our tools.

With Delta Debugging [4, 22], (visible) program behavior can be reduced to states of program variables. This technique could be used to refine path conditions even for the case that a program trace is unavailable.Therefore, we will try to combine these techniques for further investigation.

While this work is still in progress, we expect to apply it to a broader range of case studies. In particular we hope to obtain commercial safety-critical C programs, and we will extend this approach to Java, which becomes more and more important, especially as nearly every mobile phone contains a virtual machine nowadays and the threat of circulating malicious code soars.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[3] Z. Chen and B. Xu. Dependence analysis based dynamic slicing for debugging. In *Proc. ISES'01: Int. Soft. Eng. Symposium*, Wuhan, China, 2001.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, NY, USA, 2005. ACM Press.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[7] T. Gyimóthy, A. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 303–321, London, UK, 1999. Springer-Verlag.

[8] C. Hammer and G. Snelting. An improved slicer for java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22. ACM Press, 2004.

[9] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–47, New York, NY, USA, 2005. ACM Press.

[10] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[11] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, July 2003.

[12] J. Krinke. Program slicing. In *Handbook of Software Engineering and Knowledge Engineering*, volume 3: Recent Advances. World Scientific Publishing, 2005.

[13] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, pages 661–675, November/December 1998. Special issue on Program Slicing.

[14] T. Robschink. *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Universität Passau, Januar 2005. in German.

[15] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proceedings International ACM/IEEE Conference on Software Engineering (ICSE'02)*, pages 478–488, Orlando, FL, May 2002.

[16] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*. to appear 2006.

[17] F. Tip. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of the Sixth International Joint Conference on the Theory and Practice of Software Development*. LNCS 915, 1995.

[18] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[19] G. A. Venkatesh. Experimental results from dynamic slicing of c programs. *ACM Trans. Program. Lang. Syst.*, 17(2):197–216, 1995.

[20] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 512–521, Washington, DC, USA, 2004. IEEE Computer Society.

[21] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[22] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[23] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 94–106, New York, NY, USA, 2004. ACM Press.

[24] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[25] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Trans. Program. Lang. Syst.*, 27(4):631–661, 2005.