Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis

Omar Chebaro[1,2] Nikolai Kosmatov[1] Alain Giorgetti[2,3] Jacques Julliand[2]
[1]CEA, LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette France
[2]LIFC, University of Franche-Comté, 25030 Besançon France
[3]INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France
[1]firstname.lastname@cea.fr, [2]firstname.lastname@lifc.univ-fcomte.fr

ABSTRACT

Recent research proposed efficient methods for software verification combining static and dynamic analysis, where static analysis reports possible runtime errors (some of which may be false alarms) and test generation confirms or rejects them. However, test generation may time out on real-sized programs before confirming some alarms as real bugs or rejecting some others as unreachable.

To overcome this problem, we propose to reduce the source code by program slicing before test generation. This paper presents new optimized and adaptive usages of program slicing, provides underlying theoretical results and the algorithm these usages rely on. The method is implemented in a tool prototype called SANTE (Static ANalysis and TEsting). Our experiments show that our method with program slicing outperforms previous combinations of static and dynamic analysis. Moreover, simplifying the program makes it easier to analyze detected errors and remaining alarms.

Keywords: static analysis, program slicing, all-paths test generation, runtime errors, alarm-guided test generation.

1. INTRODUCTION

Recent research showed that static and dynamic analyses have complementary strengths and weaknesses, and combining them may provide new efficient methods for software verification.

The method SANTE (Static ANalysis and TEsting) introduced in [7] uses value analysis to report alarms of possible runtime errors (some of which may be false alarms), and structural test generation to confirm or to reject them. Unfortunately, in practice, when applied to real-sized programs, the method of [7] can time out leaving some alarms unknown, i.e. neither confirmed nor rejected. The experiments showed that test generation on the complete program may lose a lot of time trying to cover program paths or sections of code that are not relevant to the alarms.

The main motivation of this work is to overcome this problem in order to confirm/reject more alarms in a given time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

A recent tool demo paper [8] mentions two simple ways to integrate program slicing into the SANTE method in order to simplify and reduce the source code before costly test generation. In this paper we thoroughly investigate and evaluate smarter usages of program slicing to improve the method. We develop necessary theory on alarm dependencies and use it to determine a better synergy of the techniques. We mainly consider the class of programs supposed to terminate within a given time.

Another important motivation of this work is to automatically provide the validation engineer with as much information as possible on each detected error. For example, the error can be illustrated on a simpler program, with a shorter program path, a smaller constraint set at the erroneous statement, giving values for useful variables only, etc. Most modern verification tools do not provide such information which can considerably reduce time of analysis and correction of the error by a software developer.

We implement our method using FRAMA-C [14, 10], an open-source framework for static analysis of C code, and PATHCRAWLER [29, 4, 20], a structural test generation tool.

Contributions. The contributions of this paper include:
(1) new optimized and adaptive usages of program slicing,
(2) algorithm and implementation for these new usages,
(3) definition of a minimal slicing-induced cover,
(4) proof of underlying theoretical results,
(5) experimental results on real-life programs,
(6) detailed presentation of the extended SANTE method using value analysis, program slicing and test generation.

The short papers [7, 8] briefly described earlier versions of SANTE, respectively, without program slicing, and with the basic slicing usages (*all* and *each*) without evaluation. The advanced usages (*min* and *smart*), the underlying theoretical results and algorithm, the evaluation and comparison of all options with experiments on several real-life programs as well as the detailed presentation of the method are new.

The paper is organized as follows. Sec. 2 provides necessary background. Sec. 3 describes our method with various usages of program slicing, underlying theory and implementation issues. Sec. 4, 5 and 6 respectively provide our experiments, related work and conclusion.

2. PRELIMINARIES

2.1 Threatening statements, alarms and bugs

A *threat* is a potential runtime error provoked by the execution of some statement of a given program. Such a statement is called a *threatening statement*. There are various kinds of threats, for instance, division by 0, out-of-bound array access, invalid pointers. In the present work, we only consider two kinds of threats directly treated by our dynamic analysis tool, namely, division by 0 and out-of-bound array access. Sometimes invalid pointer errors may be treated as out-of-bound array access, but the general case of invalid pointer errors is not considered here.

Value analysis is one of the existing techniques to detect threats. Based on abstract interpretation [9], it starts from an entry point specified by the user in the analyzed program, unrolls function calls and loops, and computes over-approximated sets of possible values for the program variables at each statement. Then it uses these values to prove the absence of some threats and to report some others as possible. When the risk of a runtime error cannot be excluded, value analysis reports a threat. Such a threat detected and reported by value analysis will be called an *alarm*. When the risk of a runtime error is excluded, no alarm is reported. Essentially, an alarm is a pair containing the threatening statement and the potential error condition.

The value analysis plugin of Frama-C identifies the threatening statement and marks it by a special annotation representing the alarm. Informally speaking, for instance, for the statement `x=y/z;`, the plugin emits "*Alarm:* `z` *may be 0!*" if 0 is contained in the superset of values computed for `z`. For the last statement in `int t[10]; ... t[n]=15;` the plugin emits "*Alarm:* `t+n` *may be invalid!*" when it cannot exclude the risk of out-of-bound index `n`.

Some of the detected threats may not appear at runtime because of the over-approximation. An alarm that cannot occur at runtime is called a *false alarm*. An *error*, or a *bug*, in a program p is a threat for which there exist some inputs for p that activate the corresponding threatening statement and confirm the threat. Notice that an erroneous behavior does not necessarily result in a program crash, when for instance an out-of-bound array access leads by chance to another accessible user memory location, but we still consider such cases as bugs.

2.2 Dependence-based program slicing

Program slicing [28] is a program transformation technique for extracting an executable subprogram, called *a slice*, from a larger program. A slice has, in a certain sense, the same behavior as the original program with respect to the *slicing criterion*. A classical slicing criterion is a pair composed of a statement and a set of program variables. The slicing plugin of Frama-C accepts various kinds of other slicing criteria, e.g., a set of statements. *Dependence-based program slicing* is based on dependency analysis which includes computation of the program dependence graph (PDG) [13] showing dependence relations between program statements, and interprocedural dependency analysis allowing to deal with function calls. The Frama-C slicing plugin provides an implementation of dependence-based slicing.

Two different kinds of dependencies are distinguished: data and control dependencies (see e.g. [1]). Let us denote by $\rightsquigarrow$ the reflexive-transitive closure of the relation of data or control dependency. In other words, $l_1 \rightsquigarrow l_2$ if $l_1 = l_2$, or if the execution of l_2 depends (directly or via intermediate statements) on the execution of l_1. This relation is not necessarily symmetric: we may have $l_1 \rightsquigarrow l_2$ without $l_2 \rightsquigarrow l_1$. For instance for lines 6 and 7 of Fig. 1, we have $6 \rightsquigarrow 7$ but $7 \not\rightsquigarrow 6$.

```
0   int hasPassed(int *grades, int n){
1     int i, pass = 1, sum = 0, average;
2     for(i=0; i<=n; i++)
3       if(grades[i]<7)           // alarm1
4         pass = 0;
5     for(i=0; i<=n; i++)
6       sum = sum + grades[i];    // alarm2
7     average = sum/n;            // alarm3
8     if(average < 10)
9       pass = 0;
10    return pass;}
```

Figure 1: Example: the function `hasPassed`

We denote by *labels*(p) the set of labels of statements of a program p. Let L be a subset of *labels*(p). Basically, in dependence-based slicing techniques (see e.g. [25, Sec. 2.2] and [1, Def. 4.6]), the *slice of p with respect to L*, denoted *slice*(p, L) or p_L, is defined as the subprogram of p containing the following statements

$$\mathit{labels}(p_L) = \{\, l \in \mathit{labels}(p) \mid \exists\, l' \in L,\ l \rightsquigarrow l' \,\}. \qquad (1)$$

For a singleton $L = \{l\}$, the slice p_L is also denoted p_l. Notice that since $\rightsquigarrow$ is reflexive, $L \subseteq \mathit{labels}(p_L)$ and $l \in \mathit{labels}(p_l)$.

3. THE SANTE METHOD

This section explains how the sante method (see Fig. 2) combines *value analysis*, *program slicing* and *dynamic analysis* for C program debugging. We illustrate it on the example of the function `hasPassed` presented in Fig. 1. Given the list of grades of a student and their number, this function determines whether the student passes or fails the semester. If any grade is less than 7, or the average is less than 10, the student fails, otherwise he/she passes.

The inputs of sante are a C program p and its *precondition* which defines value ranges for acceptable inputs of p and relationships between them. For instance we define the precondition for the function `hasPassed` as:

$$\texttt{n} \geqslant 0 \ \text{ and } \texttt{grades} \text{ contains } \texttt{n} \text{ integers between 0 and 20.} \qquad (2)$$

At the first step (see Fig. 2), the value analysis proves the absence of errors for some potential threats and computes a set of alarms $A = \mathit{alarms}(p)$ reporting the remaining threats. We assume $A \neq \emptyset$ (otherwise all threats are safe). Figures 3 and 5 illustrate the Slice & Test step with different options detailed in Sec. 3.3 and 3.6. Basically, the Slice & Test options determine which and how many simplified programs should be generated and sent to dynamic analysis. According to the given Slice & Test option and the structure of dependencies in A, the slicing step produces one or several simplified versions of p, each of them containing a subset of alarms that can be triggered. For advanced options, dependency analysis is explicitly called first, otherwise it is called by the first dependence-based program slicing.

Finally, for each simplified program, dynamic analysis (detailed in Sec. 3.2) tries to activate each potential threat. This step produces for each alarm a *diagnostic* that can be `safe` for a false alarm, `bug` for an effective bug confirmed by some input state, or `unknown` if it does not know whether this alarm is an effective error or not.

3.1 Value analysis

The exhaustive list of potential threats in a given program p (with a precondition) includes all statements containing a potentially risky operation such as a division or an array

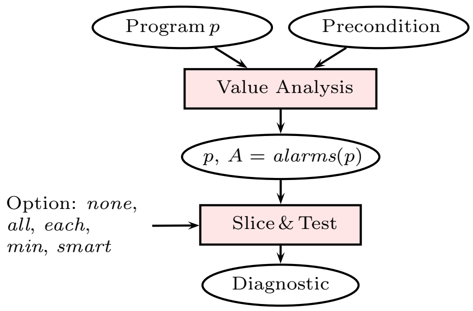


Figure 2: Overview of the SANTE method

access. The absence of errors for some of them can be established statically as explained in Sec. 2.1. Therefore, SANTE starts by applying value analysis (VA) to eliminate as many potential threats as possible. Our implementation uses the VA plugin [6] of FRAMA-C.

Each statement receives in FRAMA-C a unique implicit label even when there is no explicit label in the C code. For convenience, we identify the statement with this unique label. We denote by l_a (the label of) the threatening statement of alarm a and, in our examples, by k (the label of) the statement at line k. An alarm a is seen as a pair (l_a, c_a) containing the threatening statement l_a and the potential error condition c_a of a. This condition contains variables referenced (read) at the threatening statement l_a. The error reported by the alarm a occurs at l_a if and only if c_a is satisfied just before the execution of l_a. Notice that for both kinds of threats considered in this paper (division by 0 and out-of-bound array access), the presence of an error is determined by the values of variables referenced at the threatening statement l_a.

It will be convenient to assume that each statement is the threatening statement for at most one alarm. It simplifies the argument without lack of generality: one can either replace several alarms $(l, c_1), (l, c_2), \ldots, (l, c_k)$ on the same threatening statement by the alarm $(l,\ c_1 \vee c_2 \vee \cdots \vee c_k)$, or replace each complex statement by several simpler ones with at most one potential threat using auxiliary variables.

For the program of Fig. 1, value analysis returns the set $A = alarms(\texttt{hasPassed})$ containing the three alarms $a_1 = (3,\ \texttt{i} < 0 \vee \texttt{i} \geqslant \texttt{n})$, $a_2 = (6,\ \texttt{i} < 0 \vee \texttt{i} \geqslant \texttt{n})$ and $a_3 = (7,\ \texttt{n} = 0)$. Notice that all the three are bugs since the index `i` may be out-of-bound (equal to `n`) at lines 3 and 6, and $\texttt{n} = 0$ allowed by the precondition (2) is possible at line 7.

3.2 Dynamic analysis

Let us first define a dynamic analysis function DA.

DEFINITION 1. *Let p be a program and A be a set of alarms present in p. The dynamic analysis function DA applied to p computes a diagnostic function on A which associates to each alarm $a \in A$ one of the following results:*
1. *a pair (bug, s) for some state s, that means that an error for a occurs in p when executed on the input state s,*
2. safe*, that means that there is no error in p for a,*
3. $\mathsf{unknown}$*, that means that we do not know if there is one.*

We say that an alarm *is classified* if its diagnostic is bug or safe. In particular, the function DA returns (bug, s) for an alarm $a = (l_a, c_a)$ if and only if there is an execution path $(l_1, s_1), \ldots, (l_k, s_k), \ldots$, where s_i is the state before the execution of l_i, $s_1 = s$, $l_k = l_a$ and the error condition c_a is satisfied on the state s_k, that is, the error reported by a really occurs at the execution of l_a on s_k.

A possible implementation of DA uses the so-called *concolic* all-paths testing (see e.g. [19]). The chosen tool must guarantee that when test generation terminates normally and does not cover some program path, there exists no input state executing this path. (It is not true for tools that, unlike PATHCRAWLER, approximate path constraints.)

Technically, in order to force test generation to activate potential errors on each feasible program path in p, we add special *error branches* into the source code of p in the following way. For each alarm $a = (l_a, c_a)$, the threatening statement l_a, say `threatStatement;` is replaced by the following branching statement:

```
if( c_a ) error(); else threatStatement;
```

Test generation is then executed for the resulting C program denoted p'. We call this technique *alarm-guided test generation*. If the errror condition is verified in p', i.e. a runtime error can occur in p, the function `error()` reports the error and stops the execution of the current test case. If there is no risk of runtime error, the execution continues normally and p' behaves exactly as p. If all-paths test generation on p' terminates without covering some program path, there is no input state executing this path in p.

In our implementation, we use the PATHCRAWLER tool [4] which generates tests for all-paths criterion, or for the k-path criterion, restricting the generation to the paths with at most k consecutive iterations of each loop. Its method is similar to the *concolic testing,* also called *dynamic symbolic execution.* The user provides the C source code of the function under test. The generator explores program paths in a depth-first search using symbolic and concrete execution. The transformation of p into p' adds new branches for error and error-free states so that PATHCRAWLER algorithm will automatically try to cover error states. For an alarm a, PATHCRAWLER may confirm it as a bug when it finds an input state and an error path leading to the bug. PATHCRAWLER may also prove that the alarm is safe when all-paths test generation on p' terminates without activating the corresponding threat. When all-paths test generation on p' does not terminate, or when an incomplete test coverage criterion was used (e.g. k-path), no alarm is classified safe. Finally, all alarms that are not classified as bug or safe remain $\mathsf{unknown}$.

DA will be often applied to several slices p_j of p, returning a diagnostic $Diagnostic^j$ for the alarms of p present in p_j. Then the final $Diagnostic$ for an alarm $a \in A$ is defined as safe (resp. (bug, s)) if at least one $Diagnostic^j$ classifies a as safe (resp. (bug, s)), otherwise it is set to $\mathsf{unknown}$.

3.3 Basic Slice & Test options

In this section we present the basic Slice & Test options: *none*, *all* and *each*. Let A be the set of alarms of p.

Option *none*: The program p is directly analyzed by dynamic analysis without any simplification by program slicing. The earlier version of the SANTE method presented in [7] was limited to this unique option. Its main drawback is that dynamic analysis on a large non-simplified program may take much time or not terminate, leaving a lot of alarms $\mathsf{unknown}$.

Option *all*: In this option presented in Fig. 3a, program slicing is applied once and the slicing criterion is the set A of all alarms of p. Then dynamic analysis is applied to p_A.

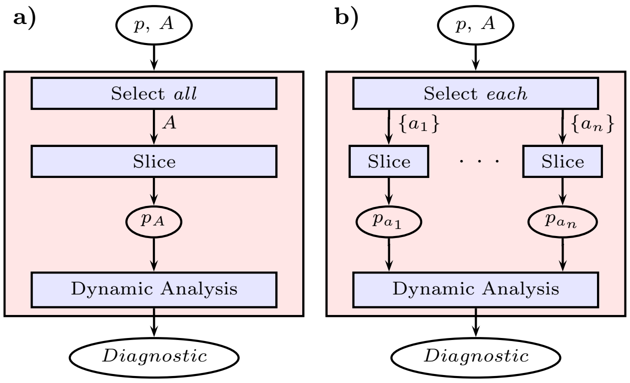


Figure 3: Basic Slice & Test options: a) *all*, b) *each*

The advantages of this option are clear. We obtain one simplified program p_A containing the same threats as the original program p. The slicing operation is executed only once. Dynamic analysis is executed only once and runs faster than for p since it is applied to its simplified version p_A.

However, since the program p_A contains all alarms present in A, dynamic analysis may time out because some alarms may be complex or difficult to analyze. In this case, alarms which are easier to classify are penalized by the analysis of other, more complex alarms, and finally many alarms may remain `unknown`. To address this drawback, we introduce the option *each*.

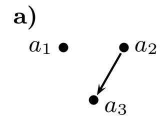
Option *each*: Assume $A = \{a_1, a_2, \ldots, a_n\}$. In this option (see Fig. 3b), program slicing is performed n times, producing a simplified program p_{a_i} with respect to each alarm a_i. Then dynamic analysis is called to analyze the n resulting programs p_{a_i}.

The advantage of this option is producing for each alarm a_i the minimal slice p_{a_i} preserving the threatening statement of a_i. Therefore, each alarm is analyzed (as much as possible) separately by dynamic analysis, so no alarm remains arbitrarily penalized by another one. Dynamic analysis for each slice p_{a_i} runs faster than for p and has more chance to classify a_i within a given time.

Among the drawbacks of this option, notice first that program slicing is executed n times and dynamic analysis is executed for n programs. Moreover, one slice may include or be identical to another one. In these cases, dynamic analysis for some of the p_{a_i} is waste of time. This is due to (mutual) dependencies between threats. We study these dependencies in Sec. 3.4 and take advantage of them in additional Slice & Test options in Sec. 3.6.

3.4 Threat dependencies

The results of this section hold for the whole set of all alarms of p and for any of its subsets. Let $A \subseteq alarms(p)$ be a set of alarms of p. Recall that an alarm a is seen as a pair (l_a, c_a) containing the threatening statement and the potential error condition of a. We say that an alarm $a' \in A$ depends on another alarm $a \in A$ if $l_a \rightsquigarrow l_{a'}$, i.e. the threatening statement $l_{a'}$ of a' depends on the threatening statement l_a of a, and we also write $a \rightsquigarrow a'$. The *program slice with respect to an alarm* a is defined as the slice with respect to the threatening statement l_a of a, and we write $p_a = p_{l_a}$. Similarly, the *program slice with respect to a set of alarms* A is defined as the slice with respect to the set of threatening statements of A, i.e. $p_A = slice(p, \{l_a \,|\, a \in A\})$.



b)

Slicing criteria for each option	
all:	$\{a_1, a_2, a_3\}$
each:	$\{a_1\}, \{a_2\}, \{a_3\}$
min:	$A_0 = \{a_1\}, A_1 = \{a_2, a_3\}$
smart:	$A_1^0 = \{a_1\}, A_2^0 = \{a_2, a_3\}$
and if a_2 still `unknown`, $A_1^1 = \{a_2\}$	

Figure 4: Slice & Test step for function `hasPassed`: a) alarm dependencies, b) slicing criteria.

We assumed that each statement is the threatening statement for at most one alarm. So, for simplicity of notation, when the error condition is not referred, we will identify an alarm $a \in A$ with the corresponding threatening statement l_a. We extend this convention to sets of alarms by considering them also as sets of statement labels. For instance, when $a = (l, c)$ is an alarm in A, we write $a \in A$ and $l \in A$ interchangeably, without any risk of confusion.

When two alarms $a, a' \in A$ are independent, program slicing with respect to a will eliminate a' in the slice. But in most cases, alarms are not all independent, and a may depend on some other a'. By definition (1) of a slice, the set $labels(p_a) \cap A$ contains the threatening statements of A which survive in p_a. Since $a \in labels(p_a)$, we have $A = \bigcup_{a \in A} labels(p_a) \cap A$.

Let $A' \subseteq A$. We say that the subset A' defines a *slicing-induced cover* of A if the family $(labels(p_a) \cap A \,|\, a \in A')$ is a *cover* of A, i.e. $A = \bigcup_{a \in A'} labels(p_a) \cap A$. We call such a cover $(labels(p_a) \cap A \,|\, a \in A')$ the *slicing-induced cover of A defined by* A'. In such a cover, each covering set $labels(p_a) \cap A$ is non-empty. We define the notion of an end alarm in $(A, \rightsquigarrow)$ as follows: $e \in A$ is an *end alarm in* A if for any $a \in A$ with $e \rightsquigarrow a$ we have $a \rightsquigarrow e$. In other words, an end alarm has no other outgoing dependencies than mutual ones. Since A is finite, it is easy to see that any $a \in A$ has a dependent end alarm $e \in A$ i.e. $a \rightsquigarrow e$. We denote by $ends(A)$ the set of end alarms of A.

Let us consider the relation $a \sim a'$ of mutual dependency defined as $a \rightsquigarrow a'$ and $a' \rightsquigarrow a$. It is an equivalence relation in A whose equivalence classes are maximal subsets of mutually dependent alarms in A. We denote by $\overline{a}$ the equivalence class of a. Lemma 1(a) shows that if an equivalence class contains an end alarm $e \in A$, then all its elements are end alarms. We denote by $ends(A/\!\sim)$ the set of equivalence classes of end alarms. Other useful properties of end alarms and slices are given in the following lemma.

Lemma 1. *Let $A \subseteq alarms(p)$ be a set of alarms of p.*
(a) *If e is an end alarm in A than every element a of its equivalence class $\overline{e}$ is an end alarm in A too.*
(b) *If $L \subseteq A$ and e is an end alarm in A that survives in the slice p_L, then $e \sim l$ for some $l \in L$.*
(c) *If $a \in A$ and e is an end alarm in A that survives in the slice p_a, then $e \sim a$.*
(d) *If $a \sim a'$ are two equivalent alarms in A, then $p_a = p_{a'}$.*
(e) *If $a \in A$ and $A' = labels(p_a) \cap A$, then $p_a = p_{A'}$.*

Proof. **(a)** Let e be an end alarm in A and $a \in \overline{e}$. Since $a \sim e$, we have $a \rightsquigarrow e$ and $e \rightsquigarrow a$. Suppose a has a dependent alarm $a' \in A$ i.e. $a \rightsquigarrow a'$. By transitivity, we have $e \rightsquigarrow a'$. Since e is an end alarm, $a' \rightsquigarrow e$, so by transitivity again, we have $a' \rightsquigarrow a$. It follows that a is an end alarm in A too.

(b) Let $L \subseteq A$ and e be an end alarm in A with $e \in labels(p_L)$. By definition (1) of p_L, there exists $l \in L$ such that $e \rightsquigarrow l$. Since e is an end alarm, $l \rightsquigarrow e$, so $e \sim l$ as required.

(c) Immediately follows from (b) for $L = \{a\}$.

(d) Follows from the definition (1) for slices p_a and $p_{a'}$.

(e) Follows from the definition (1) for slices p_a and $p_{A'}$. □

We can now state the main result of this section.

THEOREM 2. *Let $A \subseteq$ alarms(p) be a set of alarms of the program p. There exists a unique minimal slicing-induced cover of A. That is, there exists a subset $A' \subseteq A$ such that*
(a) *$A = \bigcup_{a \in A'}$ labels$(p_a) \cap A$, i.e. A' defines a slicing-induced cover of A,*
(b) *if some subset $A'' \subseteq A$ defines another slicing-induced cover of A, then* $\mathrm{card}(A'') \geqslant \mathrm{card}(A')$ *(i.e. minimality of the number of covering sets).*
(c) *if $A'' \subseteq A$ and (labels$(p_a) \cap A \,|\, a \in A''$) is another minimal slicing-induced cover of A, the covering sets of both covers are identical.*

PROOF. **(a)** We show first that there exists a slicing-induced cover of A. Choose one representative $e_i \in$ *ends*(A) in each equivalence class of end alarms $t_i \in$ *ends*$(A/\sim)$. Let A' be the set of these representatives, say $k = \mathrm{card}(\textit{ends}(A/\sim))$ and $A' = \{e_1, e_2, \ldots, e_k\}$. We claim that A' defines a slicing-induced cover of A. Indeed, any $a \in A$ has a dependent end alarm $e \in A$, whose equivalence class $\overline{e}$ has a representative $e_j \in A'$. Since $a \rightsquigarrow e$ and $e \rightsquigarrow e_j$, by transitivity we have $a \rightsquigarrow e_j$, hence $a \in \textit{labels}(p_{e_j}) \cap A$. It follows that $A = \bigcup_{a \in A'} \textit{labels}(p_a) \cap A$.

(b) Let us now show the minimality of the number of covering sets in the slicing-induced cover of A defined by A'. Suppose the subset $A'' \subseteq A$ defines another slicing-induced cover of A, i.e. $A = \bigcup_{a \in A''} \textit{labels}(p_a) \cap A$. For any $j \in \{1, 2, \ldots, k\}$, we can find $a_j \in A''$ such that $e_j \in \textit{labels}(p_{a_j}) \cap A$. Since $e_j \in \textit{labels}(p_{a_j})$ and e_j is an end alarm in A, we have $e_j \sim a_j$ by Lemma 1(c). In other words, $(a_1, \ldots, a_k)$ is another list of representatives for the different equivalence classes of end alarms $(\overline{e_1}, \ldots, \overline{e_k})$, hence the elements $a_1, a_2, \ldots, a_k$ are all different. We found at least k different elements $a_1, a_2, \ldots, a_k$ in A'', therefore $\mathrm{card}(A'') \geqslant k = \mathrm{card}(A')$.

(c) Finally we show the uniqueness of minimal slicing-induced cover of A. Assume $A'' \subseteq A$ and $(\textit{labels}(p_a) \cap A \,|\, a \in A'')$ is another minimal slicing-induced cover of A. The proof above showed that A'' contains a subset $\{a_1, a_2, \ldots, a_k\}$ where any a_j is another representative for the class of end alarms $\overline{e_j}$ and the a_j are all different. Applying the minimality for the minimal slicing-induced cover defined by A'' we obtain $\mathrm{card}(A') \geqslant \mathrm{card}(A'')$, so $\{a_1, a_2, \ldots, a_k\} = A''$ and $\mathrm{card}(A') = \mathrm{card}(A'')$. Since $a_j \sim e_j$, by Lemma 1(d) the covering sets of the both covers are identical

$$\textit{labels}(p_{e_j}) \cap A = \textit{labels}(p_{a_j}) \cap A$$

for any $j \in \{1, 2, \ldots, k\}$, that finishes the proof. □

3.5 Computing a minimal slicing-induced cover

Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of alarms of p. We actually proved in Th. 2 that any minimal slicing-induced cover of A is defined by a complete set of representatives of the classes of end alarms, and its covering sets are uniquely defined (up to the order). A complete set of representatives of the classes of end alarms can be found as follows.

(a) Using dependency analysis, compute (intra- and inter-procedural) dependencies for each alarm a_i, in particular, find the alarms a_j such that $a_j \rightsquigarrow a_i$. It gives the dependence graph $(A, \rightsquigarrow)$ (see the first step in Fig. 5).

(b) Identify the end alarms of $(A, \rightsquigarrow)$.

(c) Select a complete set of representatives $e_1, \ldots, e_k$ of the classes of end alarms of A.

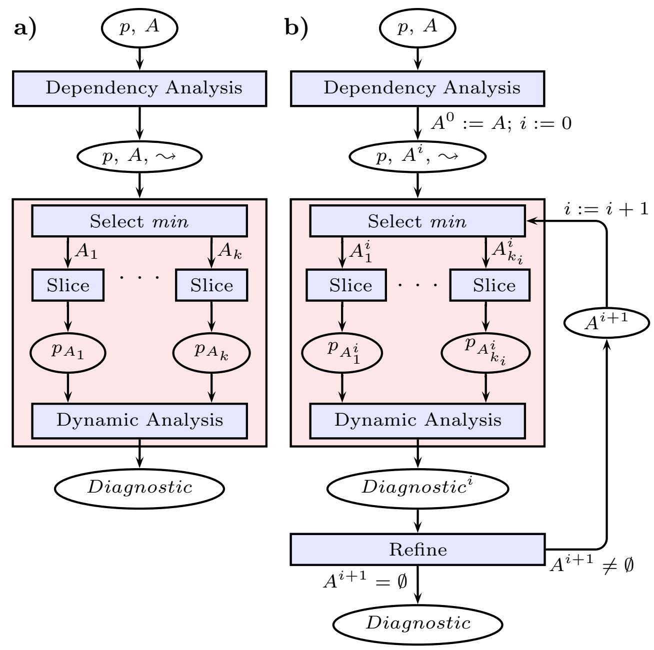


Figure 5: Advanced Slice & Test options: a) *min*, b) *smart*

Notice that step **(a)** is already included in the option *each* where program slicing for each alarm a_i calls intra- and inter-procedural dependency analysis. In practice, **(a)** is done very efficiently: in all our experiments, program slicing took less than 1 sec., while test generation took the greatest amount of time. The additional steps **(b)** and **(c)** (represented by the "Select *min*" step in Fig. 5) have only quadratic complexity in the number of alarms n. End alarms are found by definition (by examining dependencies between each alarm with each other). Representatives of the classes of end alarms can be found by a loop selecting any not-yet-marked end alarm and marking its dependent end alarms as already represented. When the graph $(A, \rightsquigarrow)$ is already available, recalculating a minimal slicing-induced cover for a subset $A' \subset A$ is also quadratic.

We are ready to show how to diminish costly calls of *DA* of the option *each* with only polynomial additional work.

3.6 Advanced Slice & Test options

This section proposes new optimized options based on alarm dependencies. Let A be the set of alarms of p.

Option *min*: This option (see Fig. 5a) calls *DA* on k slices $p_{A_1}, p_{A_2}, \ldots, p_{A_k}$ obtained by program slicing for the covering sets $A_1, A_2, \ldots, A_k$ of a minimal slicing-induced cover of A. Technically, we select a complete set of representatives $e_1, \ldots, e_k$ of end alarms of A and take the slices p_{e_i}. By Th. 2 the covering sets are $A_i = \textit{labels}(p_{e_i}) \cap A$, and we have indeed by Lemma 1(e) $p_{e_i} = p_{A_i}$. Fig. 4 shows the alarm dependencies and slicing criteria for the running example.

If all alarms are dependent then the option *min* is identical

	module *function*	*threats*	*all-threats DA* ✓	?	↯	*VA* ?	SANTE *none* ✓	?	↯	SANTE *all* ✓	?	↯	SANTE *each* ✓	?	↯	SANTE *min* ✓	?	↯	SANTE *smart* ✓	?	↯
1	libgd	15	0	14	1	12	0	11	1	0	11	1	11	0	1	11	0	1	11	0	1
	gdImageStringFTEx		TO			1s	TO			TO			1h 32m 52s			32m 16s			32m 16s		
2	Apache	12	0	9	3	12	0	9	3	0	9	3	4	5	3	0	9	3	4	5	3
	get_tag		TO			1s	TO			TO			3m 24s + 5 TO			1 TO			54s + 1 TO		
3	polygon	29	27	0	2	10	8	0	2	8	0	2	8	0	2	8	0	2	8	0	2
	main		5m 33s			<1s	1m 31s			1m 20s			7s			7s			7s		
4	rawcaudio	10	0	10	0	2	0	2	0	0	2	0	1	1	0	0	2	0	1	1	0
	adpcm_decoder		TO			<1s	TO			TO			5s + 1 TO			1 TO			5s + 1 TO		
5	eurocheck	19	18	0	1	5	4	0	1	4	0	1	4	0	1	4	0	1	4	0	1
	main		25s			<1s	18s			7s			13s			6s			6s		

Figure 6: Experimental results for *all-threats* dynamic analysis, value analysis and SANTE with different options

to *all*. If all alarms are independent then the option *min* is identical to *each*.

This option combines the advantages of the basic options *all* and *each* described in Sec 3.3. We produce simpler slices than with option *all*, hence alarms which are easier to classify are less penalized by the analysis of more complex alarms. The number of slices k is at most (and often much less than) that for *each*. Moreover, each slice p_{A_i} is important since it may be used to classify the end alarm e_i, and the analysis of p_{A_i} is never redundant.

The weakness of this option appears when the dynamic analysis of p_{A_i} times out without classifying some $a' \in A_i$, while the dynamic analysis of a potentially simpler slice (e.g. $p_{a'}$) allows to classify a'. The next option addresses this drawback.

Option *smart*: This option (see Fig. 5b) applies the *min* option iteratively on a sequence of sets of alarms A^i whose size card(A^i) decreases after each iteration. Initially, $i = 0$ and $A^0 = A$.

For each $i \geqslant 0$, we take the minimal slicing-induced cover $\{A^i_1, A^i_2, \ldots, A^i_{k_i}\}$ of the set A^i and produce the corresponding slices. The dynamic analysis generates $Diagnostic^i$ for A^i. Next, the Refine operation computes A^{i+1} as the set of alarms in $A^i \setminus ends(A^i)$ that remain unclassified (unknown) by $Diagnostic^i$. Notice that the end alarms are explicitly excluded, otherwise we could have $A^{i+1} = A^i$ and repeat the same step. Finally, we increment i and repeat the iteration for the new A^i until A^{i+1} becomes empty.

For example, in Fig. 4, if the dynamic analysis of $p_{A^0_2}$ does not classify a_2, only the end alarm a_3 is removed from A^0_2, and $A^1_1 = \{a_2\}$ for the next step ($i = 1$) of dynamic analysis which will be the last.

When A^{i+1} becomes empty, the final $Diagnostic$ classifies $a \in A$ as safe (resp. bug) if at least one $Diagnostic^i$ classifies a as safe (resp. bug), otherwise it remains unknown.

In this option each alarm is analyzed (as much as possible) separately by dynamic analysis, and it is done *exactly when necessary,* i.e. when it cannot be classified by the dynamic analysis of a larger slice. It avoids the redundancy of *each* and repairs the drawback of *min*.

4. EXPERIMENTS AND DISCUSSION

In this section, we provide experiments for different options of SANTE and compare them with one another and with a dynamic analysis technique that we call *all-threats*. This technique runs dynamic analysis in alarm-guided mode for the exhaustive list of all potential threats (without filtering by value analysis and slicing) and considers each threat as an alarm. We use five examples (up to several hundreds of LOC) extracted from real-life software where bugs were previously detected. All bugs are out-of-bound access or invalid pointers.

Ex. 1 and 2 come from Verisec C analysis benchmark [21]. Ex. 3 is an open-source program[1] used to calculate the area of a convex polygon from the coordinates of its vertices. Ex. 4 comes from Mediabench [22], Ex. 5 is an open-source program[2] containing a single function validating serial numbers on European bank notes. Experiments were conducted on an Intel quad core 2.40 GHz notebook with 4 GB of RAM with a timeout of ten minutes.

The columns of Fig. 6 show the example number and the results for each technique. The column *threats* gives the total number of potential threats before any analysis. The column *VA* gives the number of alarms reported by value analysis and sent to the Slice & Test step. The difference between the columns *threats* and *VA* gives the number of threats proved safe by value analysis. The column '✓' provides the number of alarms proven safe by the method. The columns '?' and '↯' respectively provide the number of remaining unclassified alarms and the number of detected bugs. The full process duration and the number of timeouts (TO) are given below the numbers. In our experiments, all known bugs are detected with each method.

SANTE vs. *all-threats DA*. Alarm-guided test generation in SANTE only treats the alarms raised by value analysis while *all-threats* dully considers all potential threats. In Ex. 5, *all-threats DA* analyzes 19 alarms, and it takes 25 seconds to find a bug and to prove that the error states are unreachable for the remaining 18 threats, while *DA* in SANTE analyzes only 5 alarms because 14 threats have been already proven safe by value analysis. Thus test generation in SANTE detects bugs faster and leaves less unknown alarms (cf Ex. 1, 4). Of course, when value analysis can't filter any threat (cf Ex. 2), SANTE can take as much time as *all-threats*.

SANTE *none* vs. SANTE *all*. *DA* in SANTE *all* analyzes one simplified program containing all the alarms, so it considers less paths and detects the bugs in less time (cf Ex. 5). In the worst case, when slicing does not simplify the program considerably, SANTE *all* can take as much time as *none* (cf Ex. 3).

SANTE *all* vs. SANTE *each*. In SANTE *each*, the minimal slice for each alarm is separately analyzed by *DA*. SANTE *each* leaves less unknown alarms (cf Ex. 1, 2, 4). It terminates in some cases where SANTE *all* times out (Ex. 1, 2). In Ex. 4 it classifies more alarms: *DA* analyzes two slices. It times out for one of them and terminates and proves the

[1]http://c.happycodings.com/Mathematics/code4.html
[2]http://freshmeat.net/projects/eurocheck

other target alarm safe. Of course, SANTE *each* can be slower than *all* (Ex. 5), and may waste time since *DA* on some slices can give no new information (cf Sec. 3.3 and 3.6).

SANTE *all*, *each* vs. SANTE *min*. In SANTE *min*, *DA* analyzes less programs than in SANTE *each*, thus it is faster (cf Ex. 1, 5). It can terminate in some cases where SANTE *all* times out (cf Ex. 1). SANTE *min* is never waste of time. However, SANTE *min* may time out while SANTE *each* terminates on some slices (cf Ex. 2, 4) because *DA* on the minimal slice for an alarm has the highest chances to classify it.

SANTE *each*, *min* vs. SANTE *smart*. SANTE *smart* acts like *min* in absence of timeouts (Ex. 1, 3, 5). If a timeout is encountered, SANTE *smart* analyzes smaller and smaller slices and stops only when it cannot classify more alarms. Thus it may take more time than *min* but provides better answers (Ex. 2, 4) and without the waste of time of *each* (Ex. 1, 2, 5). For instance, in Ex. 2 SANTE *each* analyzes 12 slices and goes through 5 timeouts, SANTE *min* analyzes one slice only and times out. For this example SANTE *smart* needs two iterations, *DA* times out on the first slice containing all 12 alarms and terminates on a smaller slice in the second iteration. SANTE *smart* finds the same results as SANTE *each* after one timeout. In some cases, SANTE *smart* may take more time than *none*, *min* or *all*, but its usage is in general preferable because *smart* continues iterations on smaller slices as long as that can allow to classify more alarms, and guarantees at the end that *DA* (with the same timeout) cannot classify more alarms on any slice.

Simpler counter-examples. Slicing in SANTE removes irrelevant code for the analyzed alarms. The counter-examples found execute significantly shorter paths on the simplified programs. In our experiments, the path length in counter-examples diminishes on average by 24%, this rate going up to 71% on some programs, especially with the option *each* and with the advanced options (*min* and *smart*).

Program reduction. The number of paths can be exponential in the program size. Hence even a slight reduction of the program by slicing before test generation is beneficial and can give better results for larger programs. The average rate of program reduction with the option *all* is around 24% and it goes up to 47% on some programs. With the options *each*, *min* and *smart*, the average rate is around 51% and goes up to 97% for some alarms in some programs.

The number of unclassified alarms with SANTE becomes smaller than for *all-threats DA* (resp. for *VA*), decreasing on average by 34% (resp. 47%) with the options *none* and *all*, by 67% (resp. 74%) with the *min* option, and by 82% (resp. 86%) with the options *each* and *smart*. For some examples this rate reaches 100% when all alarms are classified. Thus the number of remaining unclassified alarms is smaller with the advanced options and with the *each* option, but the *each* option method is more time-consuming.

Speedup. SANTE is less time-consuming than *all-threats DA*, and it may allow to avoid timeouts. The verification process gets particularly faster with the new advanced options (*min* and *smart*). The speedup average rate is around 43%, going up to 99% on some examples for the advanced options. To sum up, the advanced options appear to be more efficient and may allow to avoid a timeout.

5. RELATED WORK

Many static and dynamic analysis tools are well known and widely used in practice separately. Static analysis tools (e.g. FRAMA-C [14], Polyspace [24, 11]) are generally based on abstract interpretation [9] or predicate abstraction [17]. Dynamic analysis tools, such as PATHCRAWLER [29], DART/CUTE [26], SAGE [16] and EXE [5], automatically generate program inputs satisfying symbolic constraints collected by symbolic execution. The *all-threats DA* option used in our experiments is similar to these tools.

Recently, several papers presented combinations of static and dynamic analyses for program verification. Daikon [12] uses dynamic analysis to detect likely invariants. Check'n'-Crash [27] applies static analysis (the ESC/Java tool) that reports alarms but uses intraprocedural weakest-precondition computation rather than value analysis, so it necessitates code annotations and can have a high rate of false alarms. Next, random test generation (with JCrasher) tries to confirm the bugs. SANTE uses an interprocedural value analysis that necessitates only a precondition, and all-paths test generation, that may in addition prove some alarms unreachable. DSD Crasher [27] applies Daikon [12] to infer likely invariants before the static analysis step of Check'n'-Crash to reduce the false alarms rate. This method admits generated invariants that may be wrong and can result in proving some real bugs as safe, unlike SANTE which never reports a bug as safe.

Synergy [18], BLAST [3] and [23] combine testing and partition refinement for property checking. SANTE is relative to the Yogi tool that implements the algorithm DASH [2], initially called Synergy. In SANTE, we use value analysis whereas Yogi uses weakest precondition with template-based refinement. Both tools track down error states. They are specified as an input property in Yogi whereas in SANTE they are automatically computed by value analysis and error-branch introduction. Yogi does not use program slicing. It iteratively refines an over-approximation using information on unsatisfiable constraints from test generation. Its approach is more adapted for one error statement at a time, while SANTE can be used on several alarms simultaneously. [23] combines predicate abstraction and test generation in a refinement process, guided by the exactness of the abstraction with respect to operations of the system rather than by test generation. [15] is another implementation of [7] where some irrelevant code is excluded before *DA* for CFG connectivity reasons, that is weaker than program slicing.

Finally, to the best of our knowledge, advanced strategies for the integration of program slicing into a combination of value analysis and test generation for C program debugging were not previously studied by other authors.

6. CONCLUSION

In this paper we propose novel, optimized and adaptive strategies for the integration of program slicing into an innovative verification technique, called SANTE, combining value analysis and test generation. We provide a detailed description of the SANTE method with advanced usages of program slicing, study the properties of threat dependencies, introduce the notion of slicing-induced cover, establish and prove the underlying theoretical results and describe the algorithms. Compared to a basic usage of program slicing, our advanced strategies need only quadratic additional work in order to optimize the calls of costly dynamic analysis. We give a detailed evaluation of all slicing strategies, that have never been evaluated before, and compare them with one another.

Our experiments on real programs show that our combined method is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than test generation alone. The key objective of program slicing is to automatically remove irrelevant code, so that test generation on simplified programs runs faster (average speedup around 43%, going up to 99% on some examples for the advanced options) and leaves less unknown alarms within a given time. Moreover, an error is reported with more precise information, showing it on a simpler program, with a shorter program path, a smaller constraint set at the erroneous statement, giving values for useful variables only, etc. This is an important benefit in case of automatic model-based code generation where the developer has no deep knowledge of the resulting source code.

Future work includes experimenting the SANTE method on more examples and more research on different configurations of analysis techniques (options of value analysis and criteria of program slicing, using all-branch test generation, etc.).

Acknowledgment. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to Patrick Baudin, Bernard Botella, Pascal Cuoq (the main author of the value analysis plugin), Loïc Correnson, Bruno Marre, Anne Pacalet (the main author of the program slicing plugin) and Nicky Williams (the main author of PATHCRAWLER) for many fruitful discussions, lots of suggestions and advice. Many thanks to the anonymous referees for their helpful comments.

7. REFERENCES

[1] R. W. Barraclough, D. Binkley, S. Danicic, M. Harman, R. M. Hierons, A. Kiss, M. Laurence, and L. Ouarbya. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13):1372–1386, 2010.

[2] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, pages 3–14. ACM, 2008.

[3] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

[4] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *AST*, pages 70–78, 2009.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.

[6] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *SCAM*, pages 123–124, 2009.

[7] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Combining static analysis and test generation for C program debugging. In *TAP*, volume 6143 of *LNCS*, pages 652–666. Springer, 2010.

[8] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP*, pages 78–83, 2011.

[9] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[10] P. Cuoq and J. Signoles. Experience report: Ocaml for an industrial-strength static analysis framework. In *ICFP*, pages 281–286, 2009.

[11] A. Deutsch. On the complexity of escape analysis. In *POPL*, pages 358–371. ACM, 1997.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.

[14] Frama-C. Framework for static analysis of C programs, 2007–2011. http://frama-c.com/.

[15] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994, 2011.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *EMSOFT*, pages 207–216, 2008.

[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[18] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127. ACM, 2006.

[19] N. Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*, chapter XI: Constraint-Based Techniques for Software Testing. IGI Global, 2010.

[20] N. Kosmatov. Online version of the PathCrawler test generator, 2010–2011. http://pathcrawler-online.com/.

[21] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, pages 330–335. IEEE Computer Society, 1997.

[23] C. Pasareanu, R. Pelanek, and W. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *CAV*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.

[24] Polyspace. Software verification tool, 1994–2011. http://mathworks.com/products/polyspace/.

[25] T. W. Reps and W. Yang. The semantics of program slicing. Technical report, Univ. of Wisconsin, 1988.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272. ACM, 2005.

[27] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *TAP*, volume 4454 of *LNCS*, pages 1–16. Springer, 2007.

[28] M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE Computer Society, 1981.

[29] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.