



The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging

Omar Chebaro^{1,2}, Nikolai Kosmatov¹, Alain Giorgetti^{2,3}, and Jacques Julliand²

¹ CEA, LIST, Software Safety Laboratory, PC 94, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

² LIFC, University of Franche-Comté, 25030 Besançon Cedex France
firstname.lastname@lifc.univ-fcomte.fr

³ INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France

Abstract. This short paper presents a prototype tool called SANTE (Static ANalysis and TEsting) implementing an original method combining value analysis, program slicing and structural test generation for verification of C programs. First, value analysis is called to generate alarms when it can not guarantee the absence of errors. Then the program is reduced by program slicing. Alarm-guided test generation is then used to analyze the simplified program(s) in order to confirm or reject alarms.

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

1 Introduction

Software validation remains a crucial part in software development process. Software testing accounts for about 50% of the total cost of software development. Automated software validation is aimed at reducing this cost. The increasing demand on software validation has motivated much research and two major techniques have improved in recent years, static and dynamic analysis. They arose from different communities and evolved along parallel but separate tracks. Traditionally, they were viewed as separate domains. However, static and dynamic analysis have complementary strengths and weaknesses and combining them is of significant interest for program debugging.

This paper presents our tool called SANTE (Static ANalysis and TEsting) combining value analysis, program slicing and structural testing for the verification of C programs. In [1], we described an earlier version of the SANTE method combining value analysis and structural testing for C program debugging. The method used value analysis to report alarms of possible run-time errors (some of which may be false alarms), and test generation to confirm or to reject them. The method produced 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. Experimental results showed that the combined method is better than each technique used independently. It is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than a concolic structural testing tool used alone, or even guided by the exhaustive list of alarms for all potentially threatening statements.

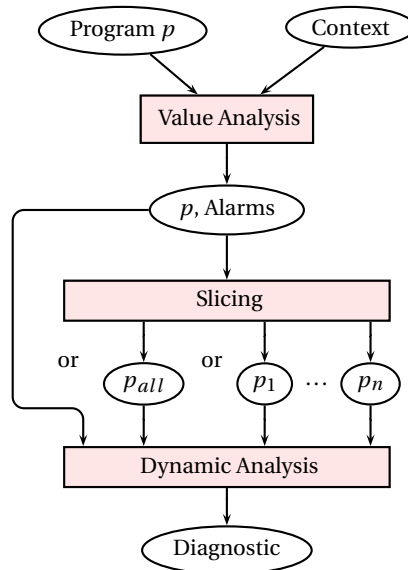


Fig. 1. SANTE Debugging Process

In the new version of the SANTE tool presented in this paper, we add program slicing to our combination in order to simplify and reduce the source code before test generation. Program slicing [2] is a technique for decomposing programs based on data and control-flow information with respect to a given slicing criterion (e.g. one or several program statements). We present two different usages of program slicing. First program slicing is performed one time with respect to the set of all alarms. Second program slicing is performed n times, once with respect to each alarm (n is the number of alarms).

Our implementation uses FRAMA-C, a framework for static analysis of C programs, and PATHCRAWLER, a structural test generation tool. FRAMA-C [3] is being developed in collaboration between CEA LIST and the ProVal project of INRIA Saclay. Its software architecture is plug-in-oriented and allows fine-grained collaboration of analysis techniques. Static analyzers are implemented as plug-ins and can collaborate with one another to examine a C program. FRAMA-C is distributed as open source with various plug-ins (i.e. value analysis, dependency analysis, program slicing, weakest precondition, ...). Developed at CEA LIST, PATHCRAWLER [4–6] is a test generation tool for C functions respecting the *all-paths criterion*, which requires to cover all feasible program paths, or the *k-path criterion*, which restricts the generation to the paths with at most k consecutive iterations of each loop.

The paper is organized as follows. Section 2 describes our tool and its implementation. Section 3 provides some perspectives and concludes.

2 The SANTE tool on a running example

This section demonstrates how, given a C program p and its execution context, the SANTE tool applies *value analysis*, *program slicing* and *dynamic analysis* for its debugging (see Fig. 1). This process is fully-automatic. The *execution context*, or *precondition*, defines value ranges for acceptable inputs of p and relationships between them. We illustrate each step of the method on the example of Fig. 2a. Given a string

```

0 int eurocheck(char *str){
1   unsigned char sum;
2   char c[9][3]={"ZQ","YP","XO",
3     "WN","VM","UL","TK","SJ","RI"};
4   unsigned char checksum[12];
5   int i = 0, len = 0;

6   if(str[0]>=97 && str[0]<=122)
7     str[0]-=32; //capitalize
8   if(str[0]<'I' || str[0]>'Z')
9     return 2; //invalid char
10  if(strlen(str) != 12)
11    return 3; //wrong length
12  len = strlen(str);

13  checksum[i]=str[i];
14  for(i=1;i<len;i++){

15    if(str[i]<48 || str[i]>57)
16      return 4; //not a digit

17    checksum[i] = str[i]-48}
18  sum=0;
19  for(i=1;i<len;i++)

20    sum+=checksum[i];
21  while(sum>9)
22    sum=((sum/10)+(sum%10));
23  for(i=0;i<9;i++)
24    if(checksum[0]==c[i][0])
25      break;
26  if(sum!=i)
27    return 5; //wrong checksum
28  return 0;} //OK

```

a) Function eurocheck

```

0 int eurocheck(char *str){
1   unsigned char sum;
2   char c[9][3]={"ZQ","YP","XO",
3     "WN","VM","UL","TK","SJ","RI"};
4   unsigned char checksum[12];
5   int i = 0, len = 0;
60  //@ assert(\valid(str+0));
6   if(str[0]>=97 && str[0]<=122)
7     str[0]-=32;
8   if(str[0]<'I' || str[0]>'Z')
9     return 2;
10  if(strlen(str) != 12)
11    return 3;
12  len = strlen(str);
130 //@ assert(\valid(str+i));
13  checksum[i]=str[i];
14  for(i=1;i<len;i++){
150  //@ assert(\valid(str+i));
15    if(str[i]<48 || str[i]>57)
16      return 4;
170  //@ assert(\valid(checksum+i));
17    checksum[i] = str[i]-48}
18  sum=0;
19  for(i=1;i<len;i++)
200  //@ assert(\valid(checksum+i));
20    sum+=checksum[i];
21  while(sum>9)
22    sum=((sum/10)+(sum%10));
23  for(i=0;i<9;i++)
24    if(checksum[0]==c[i][0])
25      break;
26  if(sum!=i)
27    return 5;
28  return 0;}

```

b) Function eurocheck with alarms

Fig. 2. Running example before and after value analysis

`str` representing the serial number of a euro banknote, this function determines whether the serial number is valid or not. Such a number normally contains one letter followed by several digits. We define the precondition for the function `eurocheck` as:

`str` is NULL or a zero-terminated string.

2.1 Step 1: Value analysis

SANTE starts by applying value analysis (see Fig. 1) to eliminate as many potential threats as possible. When the risk of a run-time error cannot be excluded by the

<pre> 0 void eurocheck(char *str){ 5 int i, len; 60 //@ assert \valid(str+0); 6 if(str[0]>=97 && str[0]<=122) 7 str[0]-=32; 8 if(str[0]<'I' str[0]>'Z') 9 return; 10 if(strlen(str) != 12) 11 return; 12 len = strlen(str); 13 for(i=1;i<len;i++){ 150 //@ assert \valid(str+i); 15 if(str[i]<48 str[i]>57) 16 return;}}</pre>	<pre> 0 void eurocheck(char *str){ 5 int i, len; 61 if(0 >= length(str)) 62 error(); 63 else 6 if(str[0]>=97 && str[0]<=122) 7 str[0]-=32; 8 if(str[0]<'I' str[0]>'Z') 9 return; 10 if(strlen(str) != 12) 11 return; 12 len = strlen(str); 13 for(i=1;i<len;i++){ 151 if(i >= length(str)) 152 error(); 153 else 15 if(str[i]<48 str[i]>57) 16 return;}}</pre>
---	---

a) The slice without error branches b) The slice with error branches

Fig. 3. The slice with respect to line 15, before and after adding error branches

(overapproximated) sets of possible values of variables for some statement, value analysis reports a threat for this statement, that is also called an *alarm*. In other words, value analysis proves the absence of errors for some potential threats and computes a set of alarms reporting the remaining threats. Our implementation uses the value analysis plug-in of FRAMA-C.

For the program of Fig. 2a, value analysis returns five alarms for (the statements at) lines 6, 13, 15, 17 and 20. At line 6, we are reading the first character `str[0]`. This alarm is a *bug* since `str` can be empty. At line 13, value analysis reports that `str[i]` may be an out-of-bound access. This alarm is a false alarm because if the length of `str` is not equal to 12, the program will return wrong length at line 11 and the execution will never reach line 13. At line 15, the alarm reported is also a false alarm. Here value analysis does not unroll all iterations, it is configured to unroll the first two iterations and then it approximates. Same for the alarms at line 17 and line 20.

Technically, the FRAMA-C value analyzer marks each alarm by an annotation printed just before it using the `assert` keyword (see Fig. 2b). For instance, at line 15, the overapproximated set of values calculated for `i` contains values greater than the length of `str` and the annotation

```
//@ assert(\valid(str+i));
```

is added just before line 15 (see line 15₀ in Fig. 2b) to report that the array access `str[i]` may be out-of-bound. The reader will find more information on the ACSL annotation language used by FRAMA-C in [3].

2.2 Step 2: Program slicing

The second step automatically simplifies the program by program slicing. In this tool demonstration, we show three different ways to simplify the program p .

1. The program p is directly analyzed by dynamic analysis without any simplification by program slicing. The earlier version of the SANTE method presented in [1] 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 unknown.
2. Program slicing is applied once and the slicing criterion is the set of all alarms of p (formally speaking, the set of threatening statements containing these alarms). We obtain one simplified program p_{all} containing the same threats as the original program p . Then dynamic analysis is applied to p_{all} (see Fig. 1). Dynamic analysis is executed only once and runs faster than for p since it is applied to its simplified version p_{all} . For the running example, p_{all} contains only 18 lines.
3. Let n be the number of alarms in p . Program slicing is performed n times, once with respect to each alarm a_i , producing simplified programs p_i ($1 \leq i \leq n$). Then dynamic analysis is called n times to analyze the n resulting programs p_i (see Fig. 1). The advantage of this option is producing for each alarm a_i the minimal slice p_i preserving the threatening statement of a_i . For the running example, we obtain five slices whose sizes vary from 3 to 16 lines. Fig. 3a shows an example of a slice for the threat in line 15.

2.3 Step 3: Dynamic analysis

Program slicing is followed by the last step, dynamic analysis, applied to all simplified programs. Dynamic analysis tries to activate each potential threat, i.e. to cover execution paths in which the associated alarms are triggered. This step produces for each alarm a diagnostic: *safe*, *bug* or *unknown*.

In our implementation, we use the PATHCRAWLER tool [5] whose method is similar to the *concolic testing* [7], also called *dynamic symbolic execution*. Given the C source code of the function under test, the generator explores program paths in a depth-first search using symbolic and concrete execution.

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, its threatening statement, say

```
threatStatement ;
```

is automatically replaced by the following branching statement:

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

where the condition determines if the error reported by the alarm occurs. For the running example, the result is shown in Fig. 3b. Test generation is then executed for

the C program with error branches denoted p' . We call this technique *alarm-guided test generation*. If the error condition is verified in p' , a run-time error can occur in p , so the function `error()` reports the error and stops the execution of the current test case. If there is no risk of run-time error, the execution continues normally and p' behaves exactly as p . 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 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 **unknown**.

For the running example, without slicing (cf Sec. 2.2.1), test generation on the original program with error branches takes around 25 seconds. When the program is sliced with respect to all alarms (cf Sec. 2.2.2), test generation finishes in around 7 seconds. For each of the five programs sliced with respect to one alarm (cf Sec. 2.2.3), test generation takes between 1 and 6 seconds. The complete time needed for the five slices is around 13 seconds. The value analysis and slicing steps are much faster than test generation (much less than 1 sec. for this example). In all cases, test generation concludes that among the five alarms, there is one bug and four false alarms.

3 Conclusion

In this demonstration paper, we presented the SANTE tool combining value analysis, program slicing and structural testing for C program debugging. The method was illustrated on a running example. Future work includes proving the soundness of the method, studying other ways to combine different analyses and transformations, and experiments on more examples.

Acknowledgments. The authors thank the members of the PathCrawler and Frama-C teams for providing the tools and support. Special thanks to Loïc Correnson and Bruno Marre for their helpful advice and fruitful suggestions.

References

1. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Combining static analysis and test generation for C program debugging. In: TAP 2010. 652–666
2. Weiser, M.: Program slicing. In: ICSE 1981. 439–449
3. Frama-C: Framework for static analysis of C programs (2007–2011) <http://www.frama-c.com/>.
4. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: EDCC 2005. 281–292
5. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST 2009. 70–78
6. Kosmatov, N.: Online version of the PathCrawler test generation tool (2010–2011) <http://pathcrawler-online.com/>.
7. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE 2005. 263–272