Limitations of static analysis tools

An evaluation of open source tools for C

Niklas Königsson

Niklas Königsson
VT 2018
Examensarbete, 15 hp
Examiner: Jerry Eriksson, Pedher Johansson
Kandidatprogrammet i datavetenskap, 180 hp

Abstract

This paper contains an evaluation of common open source static analysis tools available for C. The tools algorithms are examined and measured in a test environment designed for such benchmarks to present their strengths and weaknesses. The examined tools represent different approaches to static analysis to get a good coverage of the algorithms that are commonly used.

The test environment shows how many bugs that are correctly reported by the tools, and also how many falsely reported bug they produce. The revealed strengths and weaknesses are discussed in relation to the tools algorithms to gain a deeper understanding of their limitations.

Acknowledgements

I would like to thank my supervisors Jerry Eriksson and Pedher Johansson for their guidance in this work. The initial struggle would have been impossible without the important suggestions about direction.

Contents

1 Introduction and experiment structure

1.1 Introduction

Static analysis tools are commonly used in an iterative manner during the coding process to catch bugs early. Finding bugs late might be unnecessarily costly if extensive re-factoring throughout the code is required to remove them. Static analysis may also help members of larger projects adhere to a common code standard to increase the overall quality of the code.

This is perhaps even more important when using a relatively unrestricted language such as C. This language is still highly used, especially when a high optimization is needed. This is reflected in TIOBE programming languages popularity index(https://www.tiobe.com/tiobe-index/), which placed it as the second most popular language at the time of writing, and at the very top of the most increase in usage 2017.

This paper will evaluate some of the freely available static analysis tools available for C and find out what errors they are able to locate and, perhaps more interesting, how they are able to this. This will shed light on the current limitations among them which is the papers main purpose.

The examined tools are Splint, Clang static analyzer and Infer.

1.2 Background

In 1936, Alan Turing proved that there cannot exist an algorithm that can determine if a program terminates on any input[10]. This is the main problem faced when designing static analysis tools. The halting problem cannot be solved, but the closer we get, the more precise static analysis tools can potentially become. In this section, background work on static analysis algorithms will be presented.

1.2.1 General algorithms

The following are fundamental ways to analyze code to determine the behavior of any code. The algorithms will be frequently referenced when the chosen tools are examined in section 1.3

Pattern matching

The most straightforward way to perform static analysis is to simply match lines of code to some patterns. To do this the code is usually transformed into some canonical form to limit the amount of patterns needed. This is a very restricted method that considers very little information about the code.

One way to provide more information than a simple textual pattern matching is to make use of a syntax tree. The syntax tree is usually obtained by a compiler and can provide information about types and definitions which enable the analyzer to find bugs that require this information. For example, if the user allocates memory to a variable that requires a different size than the provided value, the syntax tree contains the relevant information for the tool to detect that there has been a type mismatch.

Flow sensitive algorithm

This is a more advanced technique that makes use of a control flow graph(CFG). The CFG is split at each conditional statement into paths that join together below it. At the merge points, the tool have rules that makes assumptions about any affected variable that can have more than one value. This rule is defined as a lattice[11] depending on the analysis being made. In some analysis it might be the wisest to assume a variables' minimal value while in other cases the maximum.

By joining the branches together, the algorithms' time complexity is kept at a very low level while still traversing and analyzing every branch. The assumptions being made at the merge points, however, will never be perfect because the analyzer has no concept of the programmers intentions. This algorithm introduces the problem of "false positives" and "false negatives". These occur when the tool reports an error that does not exist and if it misses to report an actual error.

Consider the code snippet and control flow graph depicted in figure 1. After the merge point the minimal value of x is assumed and an identical condition show that one of the subsequent paths produce a division by zero. This will cause the analyzer to alarm the user of this error even if this path is unreachable. Flow sensitive algorithms are relatively effective and can be performed in polynomial time, but because of assumptions they are not very precise.

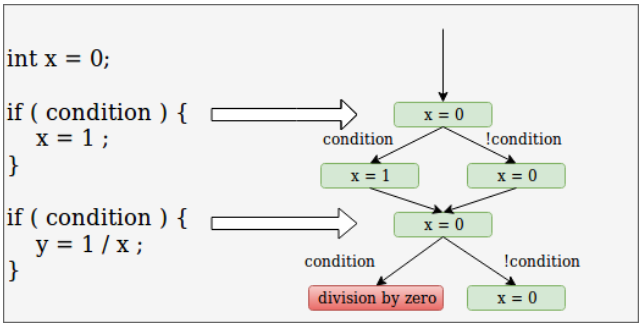

Figure 1: False positive produced when analyzing a control flow graph with a flow sensitive algorithm

Path sensitive algorithm

These algorithms also traverse a control flow graph, much like flow sensitive algorithms. But instead of merging paths after conditionals, the algorithm seeks to enumerate all possible paths in order to keep as much information as possible. This has the effect of exploding

the graph with all possible branches and also increasing the cost to exponential time with the branching factor. This is almost as costly as manually testing each branch of the program, but can be improved somewhat using other techniques like symbolic execution which is explained in the next paragraph.

Symbolic execution

This is a technique to improve the performance of path sensitive algorithms. The goal is to put constraints on the possible values of variables and in turn get closer to the halting problem[10]. When the analysis finds unknown values that are supposed to be provided in an actual execution, it uses symbols to represent them. These can be values that are provided by the user or some configuration unknown at compile time. The algorithm record constraints on these symbols and prunes the graph of impossible paths which consequently reduces false positives. Figure 2 shows how an unknown value is stored as a symbol and updated by transitions to new machine states when locating division by zero errors.

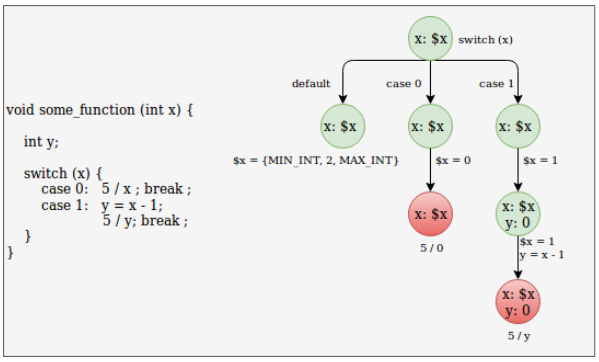

Figure 2: Recording constraints on unknown value x during symbolic execution

Context sensitivity

Algorithms with this property can reason about the input to each procedure. The algorithms records values and passes them along the control flow graph to new functions. This makes the algorithm inter-procedural, which means that it can evaluate source code globally across functions and understand each calling context. This can be local to one file or, ideally, span across entire projects. This can be achieved by using a call graph, which is a control flow graph where the nodes represent functions. the most straightforward way to analyze the call graph is to emulate functions and reanalyze each of them whenever they are called with new contexts.

Another way to achieve inter-procedural analysis is to use a summary based analysis on the call graph. In this method, each function is analyzed separately and has important information about it recorded. The analysis then uses the gathered function data to conclude important information about the code.

1.3 Algorithms behind the chosen tools

In this section, The chosen tools individual algorithms will be examined. The algorithms are to a large extent based upon work presented in the background section.

1.3.1 Splint

Overview

Splint started off as LCLint[3] in 1994, which is an analysis tool created by David Evans with time complexity as a high priority. It uses a flow sensitive algorithm that tackles the problem of false positives and negatives it produces by using specifications. These specifications were originally implemented using LCL, a specification language in the Larch family[5]. The specifications serve as a reference for the tool to look up contexts for function arguments or variables instead of a costly context sensitive algorithm. In Splints current form however, this is done by using annotations directly in the source code but the option still remains to write formal LCL specifications.

Annotations

In the article ”Static detection of dynamic memory errors”[4], Evans explains how the tool can be used to find memory errors and describes how the newly implemented annotations work. These are written before any variable in a closed comment on the form /*@[Annotation]@*/ and control the flow sensitive algorithms' assumptions about said variable at operations and merge points in the control flow graph. Figure 3 shows how annotations can be used on variables and how they must be considered in the code. This heavy dependence on annotations in the code makes it fully expected that Splint will produce a great deal of false positives when run on the test suite.

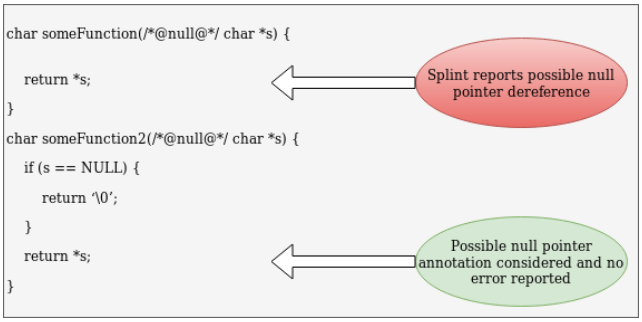

Figure 3: Splint reporting practice of a null pointer dereference in relation to an annotation

Storage model

Splint divides variables into the categories undefined, defined and completely defined. Undefined and defined are variables without or with value respectively, while completely defined describes a variable where all storage reachable from it is defined. Figure 4 shows how splint describes execution-time storage and differentiates between these classifications. An

expression that is assigned a value is used as an ”lvalue”, where only its location is of importance. Expressions used in any other way is used as an ”rvalue”, where the actual value is used. Undefined storage may not be used as an rvalue.

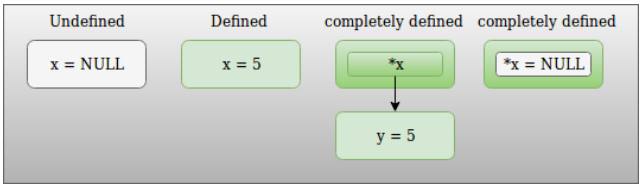

Figure 4: Splints storage model

Aliasing

A pointer is a typed memory address that can be either alive or dead. Alive pointers are either NULL or points to some storage. If this storage is undefined, the pointer is an allocated pointer while if its target is defined in any way it is an object or offset pointer depending on its target position in the struct. The pointer becomes dead if its target is deallocated. This distinction between pointers and other variables is also shown in figure 4 where storage is classified as completely defined even when the pointer is NULL. When a pointer is an object pointer, it is a member of that targets owners set. This complemented with special annotations, allows Splint to keep track of references in lack of inter-procedural capabilities and tackle aliasing problems.

1.3.2 Clang static analyzer

Overview

The Clang static analyzer is part of the Clang project(http://clang.llvm.org/). This is a compiler built using the LLVM framework(http://www.llvm.org/) designed by Chris Lattner and Vikram Adve in 2004[7]. The analyzer uses a path sensitive algorithm with symbolic execution. While constructing the control flow graph, the static analyzer employs its enabled checkers. These are meant to be implemented in a modular manner and clang static analyzer comes with many available ones as the default option. The core analyzer and its checkers creates the nodes in the graph which then saves a program point and a program state at each one of the nodes.

The program point contains information about the execution location and an abstract stack frame. The execution location consists of information regarding what the program is currently doing in the symbolic execution. This would be if it is just entering a function call or assigning a variable for example. The stack frame, on the other hand, contains useful information about the call stack that lets clang reason about inter-procedural analysis.

The program state contains four classes of information and these are environment, store, constraints and generic data map. Environment provides a mapping from clang compiler expressions to static analyzer values. Store keeps track of heap and stack memory. Constraints is the gathered data on symbolic values used in the symbolic execution. And finally the generic data map which is a place to store additional information inside the program

state. The generic data map is what checkers use to store what is needed for individual checks.

Besides pruning the graph of impossible paths with symbolic execution, the tool also reuses previous nodes if one were to be created with an already existing program point and state. This lets the tool have a very deep analysis at a reasonable time complexity cost.

Storage model and aliasing

Clang static analyzer uses a memory model developed by Zhongxing Xu, Ted Kremenek and Jian Zhang. The work is described in the article ”A Memory Model for Static Analysis of C Programs”[12]. This memory model uses clangs knowledge about program states to obtain more precise alias information and model the memory after it. This prevents multiple pointers with the same target to create new abstract representations for the same storage and brings related objects closer in the representation. This entwining of aliasing and storage removes the need for extra aliasing analysis and allows for a more informative output with fewer ”maybe” statements.

The memory is represented using a hierarchy with regions and sub-regions, who are a subsets of the stack region. The regions of memory start off as uninitialized and only obtain values the first time they are being used. Figure 5 shows how memory objects are structured in regards to hierarchy.

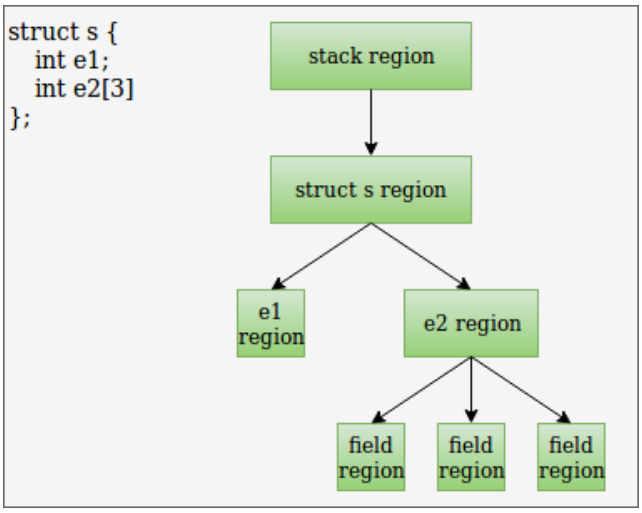

Figure 5: Clang static analyzers storage model

1.3.3 Infer

Overview

Infer is a tool acquired and developed at Facebook to handle its large code bases and daily commits. The tool started out at Monoidics based on research regarding separation logic[8] and bi abduction[1]. In 2015 the startup company was acquired by Facebook and improved upon with a combination of separation logic and abstract interpretation[2].

Infer works by first producing a call graph of procedures which are then evaluated in a summary based manner. First the procedures are sorted in topological order followed by the evaluations which has specifications about them recorded. This means that a procedure never has to be reevaluated in a new context which greatly reduces the cost. This summary based approach allows Infer to have a fully inter-procedural analysis even across separate files. The way that Infer gathers information about the procedures is depicted in Figure 6.

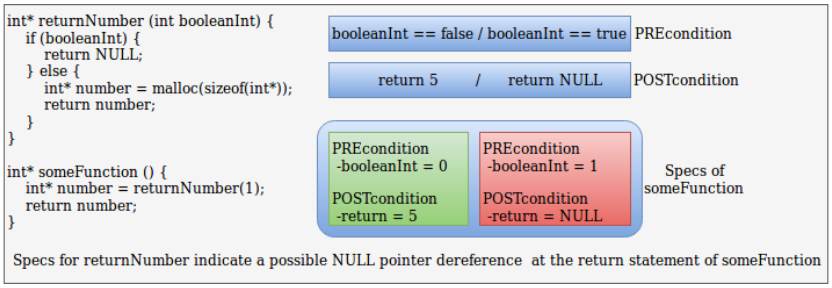

Figure 6: Infer gathering specs about a function and using it to detect a possible NULL pointer dereference

Separation logic and bi abduction

The specs gathered in Figure 6 in the previous subsection can also be explained in a more formal manner using Hoare triples, which is a way of reasoning about programs suggested by Tony Hoare in the work ”An axiomatic basis for computer programming”[6]. A Hoare triple is a representation of a procedure which can be reasoned about using representations of its pre-condition, body and post-condition.

$$\{precondition\}procedure\{postcondition\}$$

Separation logic is an extension of Hoare logic that seeks to completely separate the variables changed by an execution from the untouched ones. The most important rule in separation logic is perhaps the Frame rule which is based on the Frame problem present in artificial intelligence. The frame problem is what limitations has to be made upon the updating of variables in response to actions, or to know exactly what is changed and what remain unchanged. The Frame rule in separation logic describe that a precondition P that is affected by a procedure and produces a postcondition Q, can be disjunct with another precondition P2 which will in turn remain unchanged by the formulae.

$$\{P * P2\}procedure\{Q * P2\}$$

In the article ”A local shape analysis based on separation logic”[2], The researchers at Monoidics explains how Separation logic can be used on heap abstractions to describe them in a canonical form that can be reduced. This is another approach to symbolic execution that seeks to limit the data structure shape of symbolic heaplets and reduce them based on a set of logical rules. The rules produced by the team can prune a heaplet of junk values not needed in a computation and can even ensure program termination.

Later this was improved upon with the introduction of bi-abduction[1] which can be described as an inverse separation logic. Where the frame rule in separation logic determines

frames of unaltered states, bi-abduction infers missing anti-frames to preconditions and tries to update these during analysis. This is at the heart of Infer and allows the tool to separately analyze procedures parallel to each other to a large extent.

1.4 Method

The tools were chosen because of their different properties to gain a good coverage of the most commonly used algorithms.

- Splint(http://www.splint.org/). Flow sensitive.
- Clang static analyzer(https://clang-analyzer.llvm.org/). Flow and path sensitive, file-local context sensitive with symbolic execution.
- Infer(http://fbinfer.com/). Flow and path sensitive, context sensitive with symbolic execution and summary based analysis.

The tools will be evaluated in their ability to find bugs in the Toyota info technology centers test suite[9] for static analysis programs. The tests are hosted on the Software Assurance Reference Dataset homepage(https://samate.nist.gov/SRD/index.php), and the memory leak tests will be used as it presents a decent amount of hurdles for the programs to overcome. The test code can be viewed in Appendix A.

2 Experiment results and discussion

2.1 Results

In this section the results of the benchmark is presented. First in terms of true positives and false negatives which shows what errors were found and which ones were missed by each tool. Then in terms of false positives, which shows how many non−existent errors were reported by each tool and were. Finally the false positives reported will be described in more detail by walking through the different parts of the test.

2.1.1 True positives and false negatives

Table 1 shows all of the different errors in the test and which tools were able to locate each error correctly.

Table 1 True positives and false negatives detected by the tools in Toyota memory leak test

	Splint	Clang SA	Infer
1. Allocate in infinite loop			√
2. Double pointer to int	√		
3. Allocate in different function	√	√	√
4. Allocate inside structure	√		
5. Conditional free		√	
6. Free memory based on return value		√	
7. Switch case	√		
8. Pointer alias allocation	√	√	√
9. Dangling pointer	√	√	√
10. Two pointer alias	√	√	√
11. Pointer alias within a union		√	√
12. Union pointer struct alias	√	√	√
13. Deep union pointer alias	√	√	
14. Two double pointer alias		√	
15. Allocated memory returned	√	√	√
16. Allocated memory to global variable	√		
17. Global double pointer allocated in goto			
18. Double char pointer allocated in different function			
Total:	11	11	8

2.1.2 False positives

Table 2 shows how many false positives were reported and in which section of the test they can be found. These false positives will be described in more detail in the next subsection.

Table 2 Number of false positives reported by the tools in Toyota memory leak test

	Splint	Clang SA	Infer
1. Allocate in infinite loop			
2. Double pointer to int	1		1
3. Allocate in different function	1		1
4. Allocate inside structure		1	
5. Conditional free			
6. Free memory based on return value			
7. Switch case	3		
8. Pointer alias allocation			
9. Dangling pointer			
10. Two pointer alias			
11. Pointer alias within a union			
12. Union pointer struct alias	1		
13. Deep union pointer alias	1		1
14. Two double pointer alias			
15. Allocated memory returned			
16. Allocated memory to global variable		1	1
17. Global double pointer allocated in goto	4		1
18. Double char pointer allocated in different function	2		1
Total:	13	2	6

2.1.3 False positive details

In this section a more detailed description of the false positives will be presented. Each part of the test with falsely reported errors will be walked through in regards to the false reportings. The source code file memory_leak.c in appendix A shows the line numbers referenced.

Double pointer to int

In the second part of the test, using a double pointer, Infer reported a possible NULL pointer dereference at line 46 when accessing ptr[i]. The same possible error was also reported by Splint.

Allocate in different function

Here Splint gives a warning that the function memory_leak_003_func_001 on line 62 returns with allocated memory and a leak is possible. Infer on the other hand, warns that strcpy on line 73 might dereference a NULL-pointer with str1.

Allocate inside structure

Here Clang reported that access to the field buf on the fifth instance of memory_leak_004_s_001 results in a NULL−pointer dereference on line 96 inside strcpy.

Switch case

Here Splint reported leaks from allocations within each switch case where only one is correct. Splint also warned about the call to free on line 201 where it concluded that vptr is an unqualified argument.

Union pointer struct alias

Here Splint reported a NULL−pointer dereference on field s1 in p at line 311 when assigning a value.

Deep union pointer alias

As in the above case, Splint reported NULL−pointer dereference on field s1 in p on line 349. The same warning was also given by Infer.

Allocated memory to global variable

Clang reported a NULL−pointer is passed as an argument to strcpy on line 424 refering to memory_leak_0016_gbl_ptr. Infer reported that the same pointer could possibly be NULL.

Global double pointer allocated in goto

Splint and Infer both reported a possible NULL−pointer dereference at line 449 while indexing the global pointer memory_leak_0017_gbl_doubleptr. Splint also reported the same on lines 455 and 473 for the same pointer. Additionally, Splint also classified the pointer as unqualified storage argument when calling free on line 475.

Double char pointer allocated in different function

Both Splint and Infer warned that indexing the global pointer memory_leak_0018dst could result in a NULL−pointer dereference on line 504 and Splint also deemed it an unqualified argument to free on line 526.

2.2 Discussion

In this section the results presented in the previous section will be discussed and explained to the best of the writers ability. Each part of the test file will be discussed in the same order

as table 1. Both tables descriptions from the previous section can be used as reference.

2.2.1 Splint discussion

Splints flow sensitive algorithm relies heavily on user annotations in the code. Without these annotations the assumptions being made at the merge points of the graph are often wrong. It should be noted that Splint could perform completely without warnings if these were implemented in the test code to assign context to variables and return values.

The first thing to note in the results is that Splint is unable to handle an infinite loop. Splint can detect infinite loops if there is no value in the body that alters the value of the loop test. The loop in this case has no loop test to enable Splint to report this. What happens instead is that the loop reaches Splints upper bound and the path with the loop body is discarded completely.

Splint was also the only tool that was able to find the leak in the second test. This is because it always gives a warning when deallocating a double pointer that may have sub−allocations. In tests 5 and 6 the algorithm made bad assumptions at the merge points on lines 120 and 151 and the pointers were concluded as freed.

The abundance of warnings is what gives Splint its high true positive count, but at the same time it also gives Splint its high false positive count. This can be seen clearly in test 7 were Splint was able to find the leak but at the same time it reported the same on all switch cases.

The success on test 3 should not be interpreted as Splint having inter−procedural capabilities. The functions were evaluated on their own and a warning was given at the exit of the function at line 62 which was exited with allocated memory without annotation. Its lack of inter−procedural capabilities is also the cause of the reported double free in tests 7 and 18.

In the case of tests 11 and 14, Splint was unable to detect the allocation because the declarations at lines 274 and 372 overshadow the original allocated variable values. Splint give warnings related to the overshadowing so the score on these test might seem a bit unfair.

2.2.2 Clang static analyzer discussion

Clangs is by far the most precise of the tools with a very low false positive rate and the highest number of correct reports together with Splint. The deep path sensitive algorithm allows for an output expressed in a more confident way and the false positives still produced by Clang can be explained with weaknesses regarding symbolic execution and defines.

The Weakness regarding symbolic execution can be seen in test 4 at line 88 where the allocated variable initially have the possibility of being NULL before any conditional can reduce possible values. This causes Clang to still consider the path where s == NULL at line 91 skipping the allocation loop to produce a NULL−pointer dereference at the call to strcpy below. The second false positive produced in test 16 is expressed as a definite NULL−pointer passed as an argument to strcpy at line 424. This is because Clang does not work well with defines yet, and the tool is unable to interpret INDEX as 'a' in spite of it being defined at line 410. This causes the conditional with the allocation inside to be a definite false and no allocation is being made. Additionally, this also causes the tool to miss the leak in the test. The same weakness also causes Clang to miss the leak in test 17 were the allocating function executes depending on the define at line 460.

Clangs false negative on the first test can be explained with its implementation regarding loops. When clang reaches its upper limit on the infinite loop the path is discarded, much like the case of Splint, which causes the allocation to not be considered at all.

Clang also has trouble to differentiate between the memory locations allocated in tests 2, 4, and 18. The tool interpreters all the memory as freed after the call to free in spite of the inner allocations. This is likely an unfortunate side effect of Clangs storage model where the memory representations of the inner allocations are lost because of the hierarchy of stack regions.

2.2.3 Infer discussion

Infer has the lowest true positive rate of all the examined tools with a mediocre false positive count of 6. It is a relatively new approach to static analysis with an emphasis on being efficient in terms of time complexity which explains its low score.

The compositional shape analysis allows Infer to correctly detect the leak in the first part of the test with an infinite loop. This is possible because the tool abstracts the function into a canonical form and performs the analysis based on reduction rules. This approach seems to record the body of the loop before its upper bound is reached. This and its time complexity seems to be its only advantages at the moment however. The method fails to reduce values on tests 5, 6, 13 and 14 where Clang succeeded.

The false positives are kept at a relatively low level but some checkers are in place for additional safety regarding NULL−pointer dereferences. These activate when an initial check against NULL is absent in tests 2,3,13,16,17 and 18.

2.2.4 Conclusion

One can argue that Splint has the most severe limitations because of the heavy reliance on annotations by the coder, but these are more convenience oriented than technical. It is, however, my understanding that the use of splint has declined because of this fact. With the introduction of more automated tools, much time can be wasted by adhering to Splints strict annotation discipline. Without these the tool performs very poorly.

The obvious conclusion is that Clang static analyzer has the least limitations among the the examined tools. The priority of the developers seem to be strictly precision of error detection which is probably also the case for the common user. Clangs straightforward algorithm is also more transparent which makes the limitations easier to see in contrast to Infers more abstract approach.

Infer, on the other hand, introduces some very complex but interesting ideas to the field. The algorithm is nowhere near the performance of Clang in terms of error detection but will surely be improved upon in the coming years.

References

[1] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM (JACM)*, 58(6):26, 2011.

[2] Dino Distefano, Peter W O'hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer, 2006.

[3] David Evans. Using specifications to check source code. 1994.

[4] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Notices*, volume 31, pages 44–53. ACM, 1996.

[5] Stephen J Garland, Kevin D Jones, A Modet, and Jeannette M Wing. Larch: languages and tools for formal specification. *Springer-Verlag Texts and Monographs in Computer Science*, 1993.

[6] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[7] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[8] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

[9] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, pages 12–15. IEEE, 2015.

[10] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[11] Wikipedia. Lattice theory. https://en.wikipedia.org/wiki/Lattice_(*order*, 2018.

[12] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of c programs. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 535–548. Springer, 2010.

A Toyota−itc test suite #285

Listing A.1: memory_leak_main.c

```
1 #include "HeaderFile.h"
2
3 volatile int vflag;
4 volatile int vflag_copy;
5 volatile int vflag_file;
6
7 int idx, sink;
8 double dsink;
9 void *psink;
10
11 int main(int argc,char*argv[])
12 {
13   if(argv[1])
14   {
15
16     vflag_copy = atoi(argv[1]);
17     vflag_file = (int)floor((double)vflag_copy/1000.0);
18     vflag = (int)floor((int)vflag_copy%1000);
19     printf("vflag_file = %d vflag_func = %d vflag_copy =%d \n" ,
       vflag_file, vflag,vflag_copy);
20
21     /* Memory leak */
22     if (vflag_file ==29 || vflag_file == 888)
23     {
24       memory_leak_main();
25     }
26
27     printf("Printed from main function ");
28   }
29   else
30   {
31     printf("Enter File XXX and Function XXX \n");
32     printf("Example: To Execute 2nd File 3rd Function , Enter 002003 \n")
       ;
33     printf("Example: To Execute All Files ,Enter 888888 \n");
34     printf("Example: To Execute All functions in a File :Sample − To
       execute all functions in 3rd file,Enter 003888 \n");
35   }
36     return 0;
37 }
```

Listing A.2: memory_leak.c

```
1 /*******Software Analysis − FY2013***********/
2 /*
3 * File Name: memory_leak.c
4 * Defect Classification
5 * −−−−−−−−−−−−−−−−−−−−−−
```

```
6 * Defect Type: Resource management defects
7 * Defect Sub-type: Memory leakage
8 *
9 */
10
11 #include "HeaderFile.h"
12
13 int rand (void);
14
15 /*
16 * Types of defects: Memory Leakage - Allocate Memory and not freeing it
17 * Complexity: When using a pointer to char in an infinite for loop
18 */
19 void memory_leak_001 ()
20 {
21   int i;
22   char *buf;
23   for (i=0;;i++)
24   {
25     buf=(char*) calloc(5,sizeof(char)); /*Tool should detect this line as
       error*/ /*ERROR:Memory Leakage */
26     if(buf!=NULL)
27     {
28     buf[0]=1;
29     /* if(i>=10) */
30     }
31     /*break;*/
32   }
33 }
34
35 /*
36  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
37  * When using double pointers to the type INT
38  */
39
40 void memory_leak_002 ()
41 {
42   int **ptr = (int**) malloc(5*sizeof(int*));
43   int i,j;
44
45   for(i=0;i<5;i++)
46     ptr[i]=(int*) malloc(5*sizeof(int)); /*Tool should detect this line
      as error*/ /*ERROR:Memory Leakage */
47
48   for(i=0;i<5;i++)
49   {
50     for(j=0;j<5;j++)
51     {
52       *(*(ptr+i)+j)=i;
53     }
54   }
55   free(ptr);
56 }
57
58 /*
59  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
60  *  Memory allocated in a function and Memory used in another function
61  */
62 void memory_leak_003_func_001 (int len,char **stringPtr)
```

```
63 {
64   char * p = malloc(sizeof(char) * (len+1));
65   *stringPtr = p;
66 }
67
68 void memory_leak_003 ()
69 {
70   char *str = "This is a string";
71   char *str1;
72   memory_leak_003_func_001(strlen(str),&str1);/*Tool should detect this
       line as error*/ /*ERROR:Memory Leakage */
73   strcpy(str1,str);
74 }
75
76 /*
77  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
78  * When using pointers to the structure with an string as a member of the
       structure
79  */
80 typedef struct {
81   int a;
82   int b;
83   char *buf;
84 } memory_leak_004_s_001;
85
86 void memory_leak_004 ()
87 {
88   memory_leak_004_s_001* s=(memory_leak_004_s_001*) calloc(5,sizeof(
       memory_leak_004_s_001)) ;
89   char *s1="This is a string";
90   int i;
91   if(s!=NULL)
92   for(i= 0; i<5 ;i++)
93   {
94     (s+i)->buf = (char*)malloc(25* sizeof(char));/*Tool should detect
       this line as error*/ /*ERROR:Memory Leakage */
95   }
96   strcpy((s+4)->buf,s1);
97   for(i= 0; i<5 ;i++);
98   free(s);
99 }
100
101 /*
102  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
103  *  When using pointer to float Memory is allocated and freed in
       conditional statement(if)
104  */
105 void memory_leak_005()
106 {
107   float *ptr;
108   int flag=10;
109
110   if(flag > 0)
111   {
112     ptr= (float*) malloc(5*sizeof(float));/*Tool should detect this line
       as error*/ /*ERROR:Memory Leakage */
113     if(ptr!=NULL)
114     {
115         *(ptr+1) = 10.5;
```

```
116     }
117   }
118     if(flag < 0)
119   free(ptr);
120 }
121
122 /*
123  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
124  *  When using pointer to double Memory is allocated and freed based on
       return value of function
125  */
126 int memory_leak_006_func_001(int flag)
127 {
128    int ret;
129    if (flag ==0)
130      ret = 0;
131    else
132      ret=1;
133    return ret;
134 }
135
136 void memory_leak_006()
137 {
138   double *dptr;
139   int flag=0;
140
141   if(memory_leak_006_func_001(flag)==0)
142   {
143     dptr= (double*) malloc(5*sizeof(double));/*Tool should detect this
       line as error*/ /*ERROR:Memory Leakage */
144     if(dptr!=NULL)
145     {
146             *(dptr+1) = 10.50000;
147     }
148   }
149     if(memory_leak_006_func_001(flag) !=0)
150   free(dptr);
151 }
152
153 /*
154  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
155  *  When using Switch case statements and void pointer
156  */
157 void *vptr;
158 int memory_leak_007_func_001 (int flag)
159 {
160   switch (flag)
161   {
162     case 1:
163     {
164       vptr = (int *)calloc(10, sizeof(int));/*Tool should detect this
       line as error*/ /*ERROR:Memory Leakage */
165       if(vptr!=NULL)
166       {
167          *((int*)vptr+1) = 10;
168       }
169       return 1;
170     }
171     case 2:
```

```
172     {
173       vptr = (char *)calloc(10, sizeof(char));
174       if(vptr!=NULL)
175       {
176          *((char*)vptr+2) = 'a';
177       }
178       return 2;
179     }
180     case 3:
181     {
182       vptr = (float *)calloc(10, sizeof(float));
183       if(vptr!=NULL)
184       {
185          *((float*)vptr+3) = 5.5;
186       }
187       return 3;
188     }
189     default:
190       return -1;
191   }
192 }
193
194 void memory_leak_007 ()
195 {
196   int ret;
197   ret = memory_leak_007_func_001 (rand());
198   if(ret == 0)
199     if(vptr!=NULL)
200     {
201     free(vptr);
202     }
203 }
204
205 /*
206  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
207 *  When using 1 single pointer alias
208 */
209 void memory_leak_008 ()
210 {
211   int *ptr=(int*) malloc(5 * sizeof(int));
212   int *p = (int*)malloc(5 * sizeof(int));/*Tool should detect this line
      as error*/ /*ERROR:Memory Leakage */
213   if(ptr !=NULL)
214   {
215   p = ptr;
216   *(p+1) = 1;
217   free(ptr);
218   }
219 }
220
221 /*
222  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
223 *  When using 1 single pointer alias - dangling pointer
224 */
225 void memory_leak_009 ()
226 {
227   float *ptr=(float*) malloc(5 * sizeof(float));
228   int *p = (int*) malloc(5 * sizeof(int)); /*Tool should detect this line
      as error*/ /*ERROR:Memory Leakage */
```

```
229   if(ptr !=NULL)
230   {
231           p = (int *)ptr;
232       *(p+1) = 1.5;
233       free (ptr);
234       ptr = NULL;
235   }
236 }
237
238 /*
239  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
240 *  When using 2 single pointer alias
241 */
242 void memory_leak_0010 ()
243 {
244   int *ptr = (int*) calloc(5,sizeof(int));
245   int *p1 = (int*) calloc(5,sizeof(int));/*Tool should detect this line
      as error*/ /*ERROR:Memory Leakage */
246   int *p2 = NULL;
247   p1 = ptr;
248   p2 = p1;
249   *(p2+4) = 1;
250   free(ptr);
251 }
252
253 /*
254 * Types of defects: Memory Leakage - Allocate Memory and not freeing it
255 *  When using alias(union) - a union containing two methods of accessing
      the same data (within the same function) alias
256 */
257 typedef union {
258     char * u1;
259     char * u2;
260 } memory_leak_0011_uni_001;
261
262
263 void memory_leak_0011()
264 {
265     char * buf = NULL;
266     memory_leak_0011_uni_001 un;
267     buf = (char *)calloc(50, sizeof(char)); /*Tool should detect this
      line as error*/ /*ERROR:Memory Leakage */
268   if(buf!=NULL)
269   {
270     strcpy(buf, "This Is A String");
271     un.u1 = buf;
272   }
273     {
274       char * buf ;
275
276       buf = un.u2;
277     }
278 }
279
280 /*
281  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
282  * Complexity: Union of pointer Constant   expressions  Write
283  */
284 typedef struct {
```

```
285   int a;
286   int b;
287 } memory_leak_0012_s_001;
288
289 typedef struct {
290   int a;
291   int b;
292 } memory_leak_0012_s_002;
293
294 typedef struct {
295   int a;
296   int b;
297 } memory_leak_0012_s_003;
298
299 typedef union {
300   memory_leak_0012_s_001 s1;
301   memory_leak_0012_s_002 s2;
302   memory_leak_0012_s_003 s3;
303 } memory_leak_0012_uni_001;
304
305 void memory_leak_0012 ()
306 {
307   memory_leak_0012_uni_001 *u = (memory_leak_0012_uni_001 * )malloc(5*
        sizeof( memory_leak_0012_uni_001 ));
308   memory_leak_0012_uni_001 *p  = (memory_leak_0012_uni_001 * )malloc(5*
        sizeof( memory_leak_0012_uni_001 ));/*Tool should detect this line as
        error*/ /*ERROR:Memory Leakage */
309   p = u;
310
311   p->s1.a = 1;
312
313   free(u);
314 }
315
316 /*
317  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
318  * Complexity: Union of pointer Constant   expressions  Write
319  */
320 typedef struct {
321   int a;
322   int b;
323 } memory_leak_0013_s_001;
324
325 typedef struct {
326   int a;
327   int b;
328 } memory_leak_0013_s_002;
329
330 typedef struct {
331   int a;
332   int b;
333 } memory_leak_0013_s_003;
334
335 typedef union {
336   memory_leak_0013_s_001 *s1;
337   memory_leak_0013_s_002 *s2;
338   memory_leak_0013_s_003 *s3;
339 } memory_leak_0013_uni_001;
340
```

```
341 void memory_leak_0013 ()
342 {
343   memory_leak_0013_uni_001 *u = (memory_leak_0013_uni_001 * )malloc(5*
      sizeof( memory_leak_0013_uni_001 ));
344   if(u!=NULL)
345   {
346   u->s1 = (memory_leak_0013_s_001 *) malloc(sizeof(memory_leak_0013_s_001
      ));
347   }
348   memory_leak_0013_uni_001 *p  = (memory_leak_0013_uni_001 * )malloc(5*
      sizeof( memory_leak_0013_uni_001 )); /*Tool should detect this line as
       error*/ /*ERROR:Memory Leakage */
349   p->s1 = (memory_leak_0013_s_001 *) malloc(sizeof(memory_leak_0013_s_001
      ));
350
351   if(u!=NULL)
352   {
353   p = u;
354   p->s1->a = 1;
355     free(p->s1);
356   free(p);
357
358   }
359 }
360
361 /*
362  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
363 *  Using two double pointers to the same value within the same function
364 */
365 void memory_leak_0014 ()
366 {
367     float * fptr;
368     float **fp1 = &fptr;
369     float **fp2 = &fptr;
370     fptr = NULL;
371     {
372         float * fptr = *fp1;
373         fptr = (float *)calloc(10, sizeof(float));/*Tool should detect
      this line as error*/ /*ERROR:Memory Leakage */
374         if(fptr!=NULL)
375         {
376         *(fptr+3) = 50.5;
377         *fp1 = fptr;
378         }
379     }
380     {
381         float * fptr1 ;
382         fptr1 = *fp2;
383     }
384 }
385
386 /*
387  * Types of defects: Memory Leakage - Allocate Memory and not freeing it
388  *  Memory allocated and returned from one function and Memory used in
      another function
389  */
390 char * memory_leak_0015_func_001 (int len)
391 {
392   char *stringPtr = malloc(sizeof(char) * (len+1));
```

```
393   return stringPtr;
394 }
395
396 void memory_leak_0015 ()
397 {
398   char *str = "This is a string";
399   char *str1 = memory_leak_0015_func_001(strlen(str)); /*Tool should
        detect this line as error*/ /*ERROR:Memory Leakage */
400   if(str1!=NULL)
401   {
402       strcpy(str1,str);
403   }
404 }
405
406 /*
407 * Types of defects: Freeing a NULL pointer
408  *  Memory allocated in a function and Memory used in another function
409  */
410 # define INDEX 'a'
411 static unsigned char a =INDEX;
412 char * memory_leak_0016_gbl_ptr;
413 void memory_leak_0016_func_001 (int len)
414 {
415   memory_leak_0016_gbl_ptr=NULL;
416   if(a == INDEX)
417     memory_leak_0016_gbl_ptr= malloc(sizeof(char) * (len+1));/*Tool
        should detect this line as error*/ /*ERROR:Memory Leakage */
418 }
419
420 void memory_leak_0016 ()
421 {
422   char *str = "This is a string";
423   memory_leak_0016_func_001(strlen(str));
424   strcpy(memory_leak_0016_gbl_ptr,str);
425 }
426
427 /*
428 * Types of defects: Memory Leakage - Allocate Memory and not freeing it
429 * Complexity: When using a double pointer to long , memory allocated in
        another function inside goto label and if condition
430 */
431 long ** memory_leak_0017_gbl_doubleptr;
432 int memory_leak_0017_func_001(int flag)
433 {
434    int ret ;
435    if (flag ==0)
436      ret = 0;
437    else
438      ret=1;
439    return ret;
440 }
441
442 void memory_leak_0017_func_002()
443 {
444   int i,j;
445   memory_leak_0017_gbl_doubleptr=(long**) malloc(10*sizeof(long*));/*Tool
        should detect this line as error*/ /*ERROR:Memory Leakage */
446
447   for(i=0;i<10;i++)
```

```
448   {
449     memory_leak_0017_gbl_doubleptr[i]=(long*) malloc(10*sizeof(long));
450   }
451   for(i=0;i<10;i++)
452   {
453     for(j=0;j<10;j++)
454     {
455       memory_leak_0017_gbl_doubleptr[i][j]=i;
456     }
457   }
458 }
459
460 #define ZERO 0
461 void memory_leak_0017()
462 {
463   int flag=0,i,j;
464   memory_leak_0017_gbl_doubleptr=NULL;
465 goto label;
466
467     if(memory_leak_0017_func_001(flag)==ZERO)
468   {
469     for(i=0;i<10;i++)
470     {
471       for(j=0;j<10;j++)
472       {
473         memory_leak_0017_gbl_doubleptr[i][j] += 1;
474       }
475       free (memory_leak_0017_gbl_doubleptr[i]);
476       memory_leak_0017_gbl_doubleptr[i] = NULL;
477      }
478
479   }
480
481 label:
482       if(memory_leak_0017_func_001(flag)==ZERO)
483       {
484         memory_leak_0017_func_002();
485       }
486 }
487
488 /*
489 * Types of defects: Memory Leakage - Allocate Memory and not freeing it
490 * Complexity: When using a double pointer to char in an infinite while
      loop
491 */
492 /*Allocate Memory */
493 char **memory_leak_0018dst;
494 void memory_leak_0018_func_001()
495 {
496   memory_leak_0018dst = NULL;
497   int i;
498     {
499     while(1)
500     {
501       memory_leak_0018dst = (char**) malloc(5*sizeof(char*));
502         for(i=0;i<5;i++)
503         {
504           memory_leak_0018dst[i]=(char*) malloc(15*sizeof(char));/*Tool
      should detect this line as error*/ /*ERROR:Memory Leakage */
```

```
505             }
506             break;
507         }
508         }
509 }
510
511 void memory_leak_0018 ()
512 {
513   int i;
514     memory_leak_0018_func_001();
515     for(i=0;i<5;i++)
516     {
517       strcpy (memory_leak_0018dst[i],"STRING");
518     }
519   while(1)
520   {
521     for(i=0;i<5;i++)
522       {
523                 ;
524
525       }
526       free(memory_leak_0018dst);
527       memory_leak_0018dst = NULL;
528       break;
529   }
530 }
531
532 /*
533 * Types of defects: Memory Leakage - Allocate Memory and not freeing it
534 * Complexity:Memory Leakage main function
535 */
536 extern volatile int vflag;
537 void memory_leak_main ()
538 {
539   if (vflag == 1 || vflag ==888)
540   {
541     memory_leak_001();
542   }
543
544   if (vflag == 2 || vflag ==888)
545   {
546     memory_leak_002();
547   }
548
549   if (vflag == 3 || vflag ==888)
550   {
551     memory_leak_003();
552   }
553
554   if (vflag == 4 || vflag ==888)
555   {
556     memory_leak_004();
557   }
558
559   if (vflag == 5 || vflag ==888)
560   {
561     memory_leak_005();
562   }
563
```

```
564   if (vflag == 6 || vflag ==888)
565   {
566     memory_leak_006();
567   }
568
569   if (vflag == 7 || vflag ==888)
570   {
571     memory_leak_007();
572   }
573
574   if (vflag == 8 || vflag ==888)
575   {
576     memory_leak_008();
577   }
578
579   if (vflag == 9 || vflag ==888)
580   {
581     memory_leak_009();
582   }
583
584   if (vflag == 10 || vflag ==888)
585   {
586     memory_leak_0010();
587   }
588
589   if (vflag == 11 || vflag ==888)
590   {
591     memory_leak_0011();
592   }
593
594   if (vflag == 12 || vflag ==888)
595   {
596     memory_leak_0012();
597   }
598
599   if (vflag == 13 || vflag ==888)
600   {
601     memory_leak_0013();
602   }
603
604   if (vflag == 14 || vflag ==888)
605   {
606     memory_leak_0014();
607   }
608
609   if (vflag == 15 || vflag ==888)
610   {
611     memory_leak_0015();
612   }
613
614   if (vflag == 16 || vflag ==888)
615   {
616     memory_leak_0016();
617   }
618
619   if (vflag == 17 || vflag ==888)
620   {
621     memory_leak_0017();
622   }
```

```
623
624     if (vflag == 18 || vflag ==888)
625     {
626       memory_leak_0018();
627     }
628 }
```