Bachelor thesis
Computer Science


Radboud University

# Evaluating the performance of open source static analysis tools

*Author:*
Jonathan Moerman
s4436555

*First supervisor/assessor:*
dr. Sjaak Smetsers
s.smetsers@science.ru.nl

*Second assessor:*
Marc Schoolderman
m.schoolderman@science.ru.nl

June 24, 2018

**Abstract**

In this article, we will look at the recall, precision, and usability of five open source static analysis tools (Clang, Infer, Cppcheck, Splint and, Frama-C). Prior articles comparing the performance of static analysis tools exist, but do not include Clang and Infer and few articles have been recently published on this topic. Meanwhile, Cppcheck and Frama-C have remained in development since they were last tested. These static analysis tools are benchmarked using the test suite from Chatzieleftheriou and Katsaros [2011] as well as a new test suite introduced in this article. This new test suite addresses one of the issues found in the other test suite while also complementing it with additional scenarios. These test suites complement each other and systematically cover several kinds of defects and the situations in which they occur. Clang and Infer were found to perform favorably when compared to other well-known open source static analysis tools (Frama-C, Cppcheck, Splint).

# Contents

# Chapter 1

# Introduction

In writing non-trivial software it is unavoidable that mistakes are made. Programmers catch some of these mistakes while working on the code and some of the remaining mistakes are caught when the software is compiled or tested. Unfortunately, some defects will slip through, this is especially likely if a fault will only happen under specific circumstances or if its effects are subtle. To help find these defects static analysis tools capable of finding software defects have been made. A static analysis tool analyzes source code without executing the code.

## 1.1 Limitations of static analysis

It is not feasible to build a static analysis tool capable of finding all possible defects as not all properties of a program are decidable, and those that are are often NP-complete. For example, whether a piece of software will halt is in the general case undecidable. Alias analysis, which is often required to analyze code written in imperative languages, is another example of a problem that belongs to the class of undecidable problems [Landi, 1992]. As sub-problems of static analysis belong to the class of undecidable problems general static analysis is undecidable as well. As a result of this, a static analysis cannot be used to prove that an arbitrary piece of software does not contain defects. The duration of the analysis is an important factor for a static analysis tool as a long run time makes it infeasible to run the tool locally and slow feedback itself may be undesired. This means that a tradeoff between execution time and performance in terms of recall and precision has to be made.

## 1.2 The scope of this article

In this article, we will look at the current state of open source static analysis tools in terms of analysis performance and user experience. For a static analysis tool to perform

well its analysis should be precise, the percentage of false positives in the results should be as low as possible while the recall (the percentage of issues found) should be as high as possible. What values for the precision and recall of a tool are acceptable as well as which of the two is more important depends on the context in which a tool is used. For example, in many cases, a tool with a precision of 10% and a recall of 90% would not be acceptable. Such values may, however, be acceptable if this tool is used to help review (mission-critical) code before deployment. In this article, we will look at static analysis tools in the context of frequent/continuous use during development of software. In this case low precision is generally not acceptable as having to filter out false positives on a frequent basis would lower productivity.

### 1.2.1 Benchmarking performance

We will look at what kind of defects each tool can detect as well as look at what kind of code causes a decrease in recall and precision. We will also take a quick look at the time each of the tools takes to analyze a complex piece of software.

### 1.2.2 User experience

We will look at the following aspects of the user experience of the tested static analysis tools: description of the detected issues, ease of use and integration in the workflow of a programmer and, output filtering. For the description of the detected issues, we will look at how well the tools describe the defect and the circumstances under which this fault occurs. For ease of use, we will look at what information the user must supply to the analysis tool in order to be able to analyze a code base. For output filtering, we will take a look at whether the output can be sorted by severity and defects and what options are available for filtering out certain kinds of defects.

# Chapter 2

# Definitions

## 2.1 Test suites

A test suite is a collection of test cases made with the goal to test whether a piece of software has a certain property. In the case of this article, the test cases test whether a static analysis tool can detect real defects in a piece of code while not warning about defects in similar, but correct code.

## 2.2 Abstract domains

Static analysis tools generally use abstract states to represent the possible values of a piece of memory. For example, the possible values for an integer variable `x` may be represented as `x` $>$ `10`. The expressivity of the abstract domain to which these abstract states belong influences the quality of the analysis. Take for example the following scenario: a static analysis tool encounters the statement `x = 2*y;` while the only prior knowledge the tool had about `y` was that is an unsigned integer. If the tool is only able to represent memory states as absolute integer ranges it will be unable to learn any information from this statement. If at a later part the statement `if(x >= y){...} else {...}` is encountered the tool will be unable to decide which branch would be taken, which can lead to lower precision or lower recall. On the other hand, if the tool was able to represent the state of a variable relative to another variable it would learn enough as to know which branch would be taken.

## 2.3 Path-sensitivity and alias analysis

For a static analysis tool to perform well its path-sensitive analysis as well as its alias analysis should be sufficiently powerful. In deciding whether executing a statement can

cause a fault it is important to take the possible memory states[1] of the program at that point into account. What the state of the memory is at a given point in the program's execution may depend on how this point has been reached, which means that a static analysis will need to take into account the paths in which a statement can be reached. It is also possible that no paths leading to a statement exist, which means that this statement cannot cause a fault.

Furthermore, in imperative languages like C, it is not uncommon that two or more pointers to point to the same memory, meaning that good alias analysis (also called pointer analysis) is often vital for achieving good recall and precision.

## 2.4 Entry points

An entry point is the part of the code in which execution of a program starts, in C this is generally `int main(...)`. The availability of an entry point visible to the analysis tool provides information on the context in which a function can be called. A static analysis tool can, in that case, check from where this function can be called and limit the possible states of parameters and global variables to those feasible for these function calls. If no entry point is available the range of possible states that the memory accessible to a function can have is limited solely by the assumptions a tool makes.

## 2.5 Performance metrics

Using the number of true positives and false positives the performance of the tested tools can be compared using 3 metrics: recall, precision, and $F_{0.5}$ score.

- *Recall*, also known as sensitivity, is the number of detected true positives over the total number of positives: $Recall = \frac{TP}{P}$. A Recall of 100% means the tool has detected all true positives.
- *Precision* is the number of detected true positives over the total number of detected errors: $Precision = \frac{TP}{TP + FP}$.
- $F_{0.5}$ *score:* this measure is the F-measure ($F_\beta = \frac{(\beta^2 + 1) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$) [Chinchor, 1992] for $\beta = 0.5$: $F_{0.5} = \frac{5 \cdot Precision \cdot Recall}{Precision + 4 \cdot Recall}$. The F-measure provides a good way to take both recall and precision into account as it is strongly affected by low values for either of these measures. As false positives have a large

[1] The state of variables and other dynamic memory.

effect on the choice of the user[2] on whether or not to use a static analysis tool [Bessey et al., 2010, Johnson et al., 2013], good precision has been assigned more weight than recall. In this case, I chose a value of 0.5 for $\beta$, meaning that precision has two times the weight compared to recall when evaluating the $F_{0.5}$ score.

These measures were calculated over the tests containing the defects the tool can detect, additionally a test suite-wide precision was calculated to give insight in the amount of reported unexpected false positives.

[2]Within the context of use during development of general software.

# Chapter 3

# The static analysis tools

5 open source static analysis tools were chosen on the following grounds: The tool needs to have no dependence source code annotations. The tool must be able to detect defects that consist of incorrect or dangerous usage of functions, detecting the usage of dangerous functions does not suffice.

**Splint:** Splint (Secure Programming Lint), formerly called LCLint, is a lightweight C code analysis tool developed by the *Inexpensive Program Analysis* group at the *University of Virginia*. No new stable versions have been released since July of 2007. This tool was included as it has been used in a variety of articles[1] of similar content to this article, which could ease comparing the results from this article to other articles. The version tested is 3.1.2, the most recent stable release. Project page (mirror): `http://lclint.cs.virginia.edu/`

**Cppcheck:** Cppcheck is a static analysis tool capable of analyzing C and C++ code with simple control flow analysis. The version tested here is 1.81. Project page: `http://cppcheck.sourceforge.net/`

**Frama-C:** Frama-C is a software analysis tool co-developed at two French public institutions: *CEA LIST* software security laboratory and *INRIA Saclay - Ile-de-France* research center. It is designed to help understand what the effects and function of a given piece of code are as well as verify that source code complies with a provided formal specification. Frama-C analyzes code using plugins, most notably the *Evolved Value Analysis* plugin and the *Jessie* and *Wp* deductive verification plug-ins can be used. For this article the value analysis plugin was used as this does not require annotations (though using it can help find additional issues). Frama-C is claimed to be correct [Canet et al., 2009], meaning that this tool will never remain silent for operations in the source code which can cause a runtime error. Version 15.0 of Frama-C was used. Project site: `https://frama-c.com`

[1] Including Torri et al. [2010], Chatzieleftheriou and Katsaros [2011], Mantere et al. [2009], Zitser et al. [2004]

**Infer:** Infer is a static analysis tool developed and deployed within Facebook using separation logic and bi-abduction to reason about memory mutations. It is capable of analyzing C, C++, Objective-C and Java source code. Recent versions of Infer are able to find flaws in source code using methods other than separation logic analysis. Version 0.12.1 was used. Project site: `http://fbinfer.com`

**Clang:** The Clang static analyzer is a static analysis tool built on top of the Clang compiler front-end and LLVM capable of analyzing C, C++, and Objective-C code. Clang was not used directly, instead, Scan-Build was used, which is a small front-end utility made by the Clang developers. Version 5.0 of Clang was used. Project site: `https://clang-analyzer.llvm.org/`, scan-build utility: `https://clang-analyzer.llvm.org/scan-build.html`

To my knowledge, neither Infer or Clang has ever been compared to other static analysis tools in an article similar to this.

# Chapter 4

# Method

## 4.1 The different kind of detections

When analyzing the output of the static analysis tools detections of supposed defects are categorized into two categories and recorded. The tests from test suites used to benchmark the static analysis tools contain two very similar versions of the same code. One of these versions contains a single defect while the other does not contain any defects. We will call these versions $T_{defect}$ and $T_{correct}$ respectively. $T_{defect}$ consists of three parts $C_{pre}$, $S$ and $C_{post}$ where $C_{pre}$ is the code preceding statement $S$ and $C_{post}$ is the code following this statement. The code in $C_{pre}$ and $C_{post}$ does not contain any statements that can cause a fault, but $C_{pre}$ and $S$ combined does contain a defect. In that case, $S$ can or will cause a fault. $T_{correct}$ is semantically identical to $T_{defect}$ with $C_{pre}$ replaced with a slightly modified $C_{pre}^{correct}$ such that $T_{correct}$ does not contain any defects. If a tool detects the defect in $T_{defect}$ then this detection is categorized as a true positive (TP). The true positives are the subset of positives the tool was able to detect. All other detections are categorized as false positives (FP) unless this detection can be explained by prior detections in the same code[1] in which case it is ignored.

In this article we distinguish two types of false positives: An expected false positive (eFP) is a detection in statement $S$ of $T_{correct}$ of the same issue present in $T_{defect}$, despite this defect not existing in $T_{correct}$. All other false positives are categorized as unexpected false positives (uFP) as the test case was not designed to test for these detected issues.

An example of true positive and false positive detections can be seen in listings 4.1 and 4.2. The code in listing 4.1 contains a single defect: `dangerous_func` is called with `false` as value for both `a` and `b`, which causes a *use after free* error at line 15 (this is statement $S$). If a *use after free* error is detected for the same statement in listing 4.2 then this detection is considered to be an expected false positive as here this statement

[1] Example: Tool $A$ detects that *var* is used at line $x$ while it can be `null`, then subsequently detected null pointer issues related to the use of *var* are ignored.

Listing 4.1: A possible $T_{defect}$ test case containing a use after free defect

```
1  void dangerous_func(int* ptr, bool a, bool b) {
2      int val = 0;
3      if(!ptr) return;
4
5      if(a) {
6          *ptr += 2;
7      } else {
8          val = *ptr; /* uFP: Use of null pointer detected: ptr*/
9          free(ptr);
10     }
11
12     if(b) {
13         val += 5;
14     } else {
15         val += *ptr; /* TP: use after free detected: ptr */
16     }
17
18     if(a) free(ptr);
19
20     printf("val = %i\n", val);
21 }
22
23 int main() {
24     /* Unsafe function call */
25     dangerous_func(calloc(1, sizeof(int)), false, false);
26
27     return 0;
28 }
```

Listing 4.2: A possible $T_{correct}$ counterpart for listing 4.1

```
1  void dangerous_func(int* ptr, bool a, bool b) {
       ...
15         val += *ptr; /* eFP: use after free detected: ptr */
       ...
21 }
22
23 int main() {
24     /* Safe function call */
25     dangerous_func(calloc(1, sizeof(int)), true, false);
26
27     return 0;
28 }
```

can cause no issues as `a` is `true`. A detection of the use of a null pointer at line 8 is considered to be an unexpected false positive as $T_{defect}$ (listing 4.1) does not contain this issue.

Unexpected false positives can help highlight what kind of code is problematic for a given static analysis tool. In case of this article, some kinds of code resulted in the same false positive detections, regardless of the kind of defects present. One may not want to assign the same weight to both expected as unexpected false positives as one kind of defect could have a disproportionately large impact on the results.

## 4.2 Description of the test suites

### 4.2.1 Test suite from *Test-driving static analysis tools in search of C code vulnerabilities*

To benchmark the static analysis tools two test suites were used. The first test suite was introduced in *Test-driving static analysis tools in search of C code vulnerabilities* by Chatzieleftheriou and Katsaros [2011]. This article will from this point onwards be referred to as *C&K-2011* and the test suite will be referred to as *C&K-2011TS*. This test suite was used as it contains (synthetic) tests for a wide variety of defects in a variety of situations. This test suite contains just under 800 tests with which a tool's ability to detected the defects described in table 4.1 can be measured. For each kind of defect, the test suite contains a base case as well as 15 tests requiring path-sensitive analysis, 7 tests requiring context-sensitive[2] (taking calling context into account) analysis and 2 tests requiring alias analysis. The test suite also contains several tests for variations on defects. For these tests, the test suite only contains the most simple cases suitable only for measuring if a tool is able to detect that defect.

The path-sensitive tests, of which an example can be seen in listing 4.3, require the static analysis tools to exclude impossible paths from results while reporting about errors in the possible paths. The context-sensitive tests, of which an example can be seen in listing 4.4, require the tools to take the calling context into account when analyzing function calls. The tests requiring alias analysis require the tools to keep track of what the variables point to. The path-sensitive tests do not require the tool to take a calling context into account and for context-sensitive tests, no paths need to be excluded from analysis. Both the path-sensitive as context-sensitive tests may require alias analysis in some capacity.

The way in which this test suite is used is slightly different from *C&K-2011*. In *C&K-2011* a single version of each program was used to test both if a tool will detect a true defect (listing 4.3 line 11) as well as test if it will not yield a detection in a similar, bot correct, situation (listing 4.3 line 9). Here two different versions of the program are used,

[2]Context sensitivity can be seen as a subset of path sensitivity as previously defined.

Table 4.1: Frequent code defects as described in *table I of C&K-2011* with *File Operations: Access without open* folded into *General: Uninitialized variables* and arrays and strings combined.

| **Categories** | **Defects** | **Description** |
|---|---|---|
| General | Division by zero | Divide a value by zero (CWE-369) |
| | Null pointer dereference | Dereference a pointer that is NULL (CWE-476) |
| | Uninitialized variables | Use a variable which has not been initialized (CWE-457) |
| Integers | Overflow | An integer is incremented to a value that is too large to store in its internal representation (CWE-190) |
| | Sign errors | A signed primitive is used as unsigned value (CWE-195) |
| | Truncation errors | A primitive is cast to a primitive of smaller size (CWE-197) |
| Arrays and strings | Direct overflow | Out-of-bounds access of an array (CWE-119) |
| | Null termination errors | A string is incorrectly terminated (CWE-170) |
| | Off-by-one errors | Use a min or max array index which is 1 more or less than the correct value (CWE-193) |
| | Truncation errors | A string has been truncated and possible important information is lost (CWE-222) |
| | Unbounded copy | Copy array without checking the size (CWE-120) |
| Format string vulnerabilities | | Invalid format-string in printf-like functions (CWE-134) |
| Memory | Double free | Call free() twice in the same memory address (CWE-415) |
| | Improper allocation | Misuse of functions that allocate memory dynamically |
| | Initialization errors | Not initialize or incorrectly initialize a resource (CWE-665) |
| | Memory leak | Not release allocated memory (CWE-401) |
| | Failure check | Not check for failure of functions which are used for dynamic allocation of memory |
| | Access freed memory | Access memory after it has been freed (CWE-416) |
| File Operations | Access closed file | Access a file which has previously been closed |
| | Access in different mode | Access a file in a different mode than the one it has been specified when opening the file |
| | Double close | Close a file descriptor two times |
| | Resource leak | Not close a file descriptor (CWE-403) |
| | Failure check | Not checking for failure of functions which are used for opening a file |
| Concurrency errors | Deadlock (no multithreading) | Incorrectly manage control flow during execution (CWE-691) |
| | Deadlock (multithreading) | Insufficient locking and unlocking of a thread (CWE-667) |
| | Time Of Check, Time Of Use (TOCTOU) errors | Check the state of a resource and try to use it at a later moment based on this invalid info (CWE-367) |

Listing 4.3: A simple path sensitive example from the test suite from *C&K-2011*

```
1  int main()
2  {
3      int *z = 0;
4      int k;
5      int x;
6
7      x = 10;
8      if(x > 20) {
9          k = z[2]; /* OK */
10     } else {
11         k = z[2]; /* DANGER */
12     }
13
14     return 1;
15 }
```

one identical to the program used in *C&K-2011* and one version in which the defect is corrected. This has two advantages: It makes it easier to see which detections are related to the true positive. It could in some cases lead to more detections as some tools may abort further analysis after detecting an issue that it considers critical. Analyzing this additional version of the code can thus make analysis easier while potentially leading to a more accurate insight into the precision of a tool. The path sensitivity tests that make up the majority of the tests in the test suite from *C&K-2011* measure if the tools are able to detect which statements are executed and which never are. However, one can argue that this isn't an accurate way of measuring the precision of a tool. Some static analysis tools will detect issues in statements that are unreachable, but could under no circumstance be correct. An example can be seen in listing 4.3: the statement at line 9 can never be correct as `z` never has a different value assigned to it than 0. This statement will never be executed, yet it may still deserve some attention as it cannot be correct code. A discussion can be held on the topic of whether detecting issues in unreachable code is desirable, however, this is outside the scope of this article. Testing the path-sensitive analysis in a way that is limited to excluding dead code does not give a sufficiently accurate picture of how a tool will perform in real life scenarios.

A more realistic way to measure path-sensitivity would be to test the tools ability to detect defects in code in which for every statement a path exists in which it can be reached and executed without causing a fault in the function in which it is contained. In this case, excluding dead code from further analysis is no longer enough as now a tool needs to take the different paths in which a statement can be reached into account and not just if such a path could exist.

Listing 4.4: A context sensitive example from the test suite from *C&K-2011*

```
1  int ar[5] = {0,0,0,0,0};
2  int *z = 0;
3  int k;
4
5  void assign(int *x, int *y)
6  {
7      *x = *y;
8  }
9
10 int main()
11 {
12     assign(&k,z); /* DANGER */
13
14     assign(&k,ar); /* OK */
15
16     return 1;
17 }
```

### 4.2.2 New test suite

To test tools in a more realistic way I created a new test suite containing 300 tests, in which all statements can be reached without causing a fault[3]. Whether a piece of code can trigger a fault is determined by the path that is taken to reach the end of the code. An example of a function that can be found in this new test suite is the function `dangerous_func` in listing 4.5, this function is analogous to the code in listing 4.3 with the biggest difference being that every statement can be reached. One situation exists in which a *null pointer* is dereferenced: the situation in which line 7 is not executed and line 13 is executed ($x \geq y$, $x \leq 10$). If line 7 was executed and line 13 is executed no defect is encountered. In the example in listing 4.5, it is not possible for all lines of code to be executed during a single function call, in other tests containing loops this is possible with a part of the paths where this is the case leading to a defect. Like the suite from *C&K-2011* this test suite can be used to test how capable an analysis tool is in detecting defects in programs where an entry point is provided and the value of all function arguments can be fully determined by analyzing the code available to the tool. This new suite can also be used to test static analysis in situations where no entry point is provided as would be the case when analyzing a library instead of an application. In this case, the possible values of the arguments with which a function can be called with are not known, the tools have to detect if an input exists for which an incorrect state is reached in the function. In this case, each test case consists of a dangerous version of a function and a version of this function that aborts if a dangerous input is supplied. For these tests a number of assumptions are made:

- Parameters of value types like integers or chars can have all possible values.

[3]Not necessarily in a single function call

Listing 4.5: simple example from the new test suite

```
1  void dangerous_func(int x, int y) {
2      int arr[3] = {1, 2, 3};
3      int* ptr = 0;
4      int val;
5
6      if(x < y) {
7          ptr = arr;
8      }
9
10     if(x > 10) {
11         val = arr[2];
12     } else {
13         val = ptr[2]; /* DANGER, if x >= y & x <= 10 */
14     }
15
16     printf("x=%i: %i", x, val);
17 }
18
19 int main() {
20 #ifdef UNSAFE_INPUT
21     dangerous_func(10, 10);
22 #else
23     dangerous_func(9, 10);
24     dangerous_func(11, 10);
25 #endif
26 }
```

- Parameters that are pointers are not NULL unless the parameter is compared to NULL.
- Parameters that are pointers pointing to a resource do not point to a freed or closed resource.

This test suite will from this point onwards be referred to as *JM-2018TS*. Details on the general structure of the tests contained in this suite can be found in appendix B.

## 4.3 How the test programs were analyzed

The basic method of benchmarking the tools was the same for both test suites. For both suites, two versions of the same code are used per test case, as described earlier in section 4.1. $T_{defect}$ is used to detect if the tools are able to detect the defect, in which case a true positive is recorded. $T_{correct}$ is used to determine if the tools report issues which are in fact false positives.

Both suites contain two kinds of test cases for each kind of defect: a base case which is used to determine if a tool is able to detect a certain kind of defect and a set of test cases to benchmark the performance of finding this kind of defect under different circumstances. If analysis of the base case yields a true positive and no expected false positives[4] in the base case then the defect is considered detectable by that tool.

*JM-2018TS* contains tests for most[5] defects printed in bold in table 5.1. Every kind of defect for which this test suite contains tests receives the same amount of testing, unlike *C&K-2011TS* that included additional tests to check whether a tool is able to detect certain variations on tests.

### 4.3.1 Suite from *C&K-2011*

We will now look in more detail at how the tests from both test suites were analyzed, starting with *C&K-2011TS*. As each of the programs contained an entry point (`int main`) programs were analyzed one at a time. Except for Splint[6], all tools were bench-

[4] One of the tools issues warnings that could, at first sight, be classified as a true positive and a false positive for the base case of the same defect. In this case the tool's warnings were related to usage of a programming construct instead of incorrect usage of said construct, for example: issuing a warning when the value of a signed integer variable is assigned to an unsigned integer variable is not the same as issuing a warning when a negative value is assigned to an unsigned integer variable.

[5] Missing are tests in the *concurrency errors* category and tests that contain issues related to missing checks for operations that could fail.

[6] Splints output contained a large amount of noise, information that was of no use detecting the defects present in the suite. To ease analysis Splint was called from the command line in conjunction with the following arguments to filter out most of the warnings unrelated to the subject of the tests: `-unreachable -exportlocal -predboolint -compdef -noret -mayaliasunique -nestcomment -retvalint -globstate -nullret -dependenttrans -retvalother -varuse -compdestroy +voidabstract`.

marked from the command line using their default configuration. Analyzing part of the tests in this suite did result in detections of unintended defects[7]. In most cases the tests could be altered to fix this, however, this wasn't the case in tests where a function would return a pointer to memory that was freed within this function. Removing this property of the test could not be done without changing the way in which the test works. As this issue did interfere with the ability to detect the true defect for tools which detected this issue the results were omitted for these tools[8].

### 4.3.2 *JM-2018TS*

For the new suite, the configuration of the tools as well as the set of tools that was benchmarked was changed. Frama-C and Clang performed far worse on tests containing loops than they did on tests containing loops in *C&K-2011TS*. Meanwhile the ratio between true positives and expected false positives was reduced to just above 50% for Infer. Splint was not tested using this benchmark as Splint's analysis proved very weak when analyzing the results from *C&K-2011TS* and initial testing showed similar performance on *JM-2018TS*. Further contributing was that Splint's analysis resulted in output that is time-consuming to analyze and it failed to parse several system library headers rendering it useless in many real-world scenarios, thus the choice was made to explore configuration options for improving the performance for the other tools instead of further benchmarking Splint. Clang and Frama-C offered options to improve the analysis of loops. Infer's bad performance seemed to be caused by this tool analyzing functions for all possible parameter values, not just the parameter values possible during the execution of the program. To test this theory Infer was also benchmarked using a modified version of the test suite. In this suite, functions of the versions of the programs that do not contain any defects abort if dangerous parameters are supplied. All tools were benchmarked using their default configurations[9]. Additionally, Clang and Frama-C were benchmarked with an option to unroll loops to the number of iterations used in the test suite or allow a statement of code to be analyzed this number of times.

For the case where no entry point is provided the method used was nearly the same for most tools. For most tools, the setup was identical except that all tests within a category were analyzed at once. Here all pieces of code could be analyzed within one call to the tool as the code did not contain conflicting entry points. Frama-C did however not support analyzing code without an entry point. To overcome this issue each test case had to be processed separately with the name of the test function supplied as the

[7] A simple example would be attempting to access memory that has been allocated using malloc, but for which it hasn't been checked if the allocation was successful. In this case, adding a check to safely abort the test in case allocation was unsuccessful fixes the issue. Note that in this case, the only effect of a missing check is an additional detection while it does not influence the tool's ability to detect the true defect.

[8] The vast majority of these test cases consist of context-sensitive tests from the memory category. These tests make up a small portion of the test suite, in the worst case (Clang) the $\frac{1}{3}$ of results of some of the types of context-sensitive tests was omitted, for other tools, this was $\frac{1}{4}$ (Cppcheck) and $\frac{1}{6}$ (Frama-C)

[9] Program parameters were limited to input files and `-I` inclusion flags.

entry point. As Frama-C analyzes a program for all feasible parameters for the entry point the results should be comparable. This method proved to be very time-consuming as Frama-C had to be called twice for each test for which each call to Frama-C has to be supplied with different arguments. Instead, this method was used to get results for two categories and the issues identified were used to extrapolate results from the case where an entry point is provided[10].

When benchmarking the tools with tests which lack an entry point, Infer was no longer tested in two ways. This was no longer needed as the alternative version of the test suite is identical to this version of the test suite, with the only difference being that no entry point is provided.

[10]This may cause some false positives to be missed and recall to be not entirely accurate, yet the results should not differ much as the tool was quite consistent in its results.

# Chapter 5

# Results

## 5.1 Defects detectable by the tools

Table 5.1 shows which kind of defects were detected by which tools. Most kinds of defects could only be detected by one or two of the tools. When looking at the main categories of defects 20% of the types of defects were only detected by one of the tools, 40% by two tools. 20% of the types of defects were detected by 3 of the tools and 20% of the types of defects were detected by 4 or all of the tools. Due to differences between the test suites improper memory allocation defects were only detected in JM-2018TS. In both cases, the test programs are built around calling a function of the malloc family with a size parameter of 0, but only in *JM-2018TS* is the newly 'allocated memory' ever read from or written to. Two of the tools (Clang and Frama-C) detected usage of such memory, but the allocation itself did not yield any warnings.

Defects contained in table 5.1 printed in bold are used to measure the performance of the tools in terms of recall and precision. The other defects can be considered to be variations on these defects and are solely used to construct table 5.1.

## 5.2 General trends and differences between test suite results

Table 5.2, 5.3 and 5.5 show how the tools performed on each kind of test from the two test suites[1]. Table 5.2 shows the results for the test suite from C&K-2011 while table 5.3 shows the results for JM-2018TS for the case in which an entry point into the code is provided, finally table 5.5 shows the results for JM-2018TS for the case in which no entry point is provided. For each tool, these tables show the recall, the percentage of cases in which a true positive is accompanied by an expected false positive (**eFP/TP**) for each

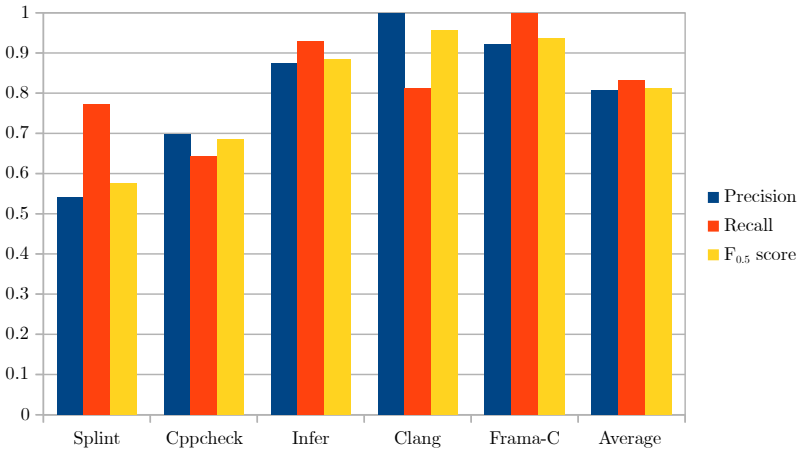
[1]Black cells denote that the value is undefined (division by zero).

Table 5.1: Tools capability to detect certain types of defects

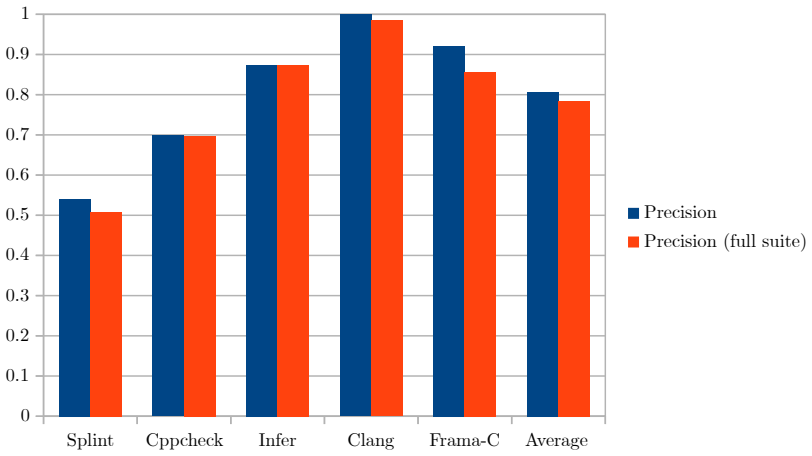
| Categories | Problems | | Splint | Cppcheck | Infer | Clang | Frama-C |
|---|---|---|---|---|---|---|---|
| General | ***Division by zero*** | | N | Y | N | Y | Y |
| | ***Null pointer dereference*** | | Y | Y | Y | Y | Y |
| | Uninitialized variables | ***Integers*** | Y | N | N | Y | Y |
| | | Strings | Y | N | N | N | Y |
| | | Arrays | Y | N | N | Y | Y |
| | | ***Pointers*** | Y | Y | N | Y | Y |
| Integers | ***Overflow*** | | N | N | N | N | Y |
| | ***Sign errors*** | | N | N | N | N | N |
| | ***Truncation errors*** | | N | N | N | N | N |
| Arrays and Strings | ***Direct overflow*** | | N | Y | N | N | Y |
| | ***Null termination errors*** | | N | N | N | N | N |
| | ***Off-by-one errors*** | | N | Y | N | N | Y |
| | ***Truncation errors*** | | N | N | N | N | N |
| | Unbounded Copy | ***memcpy*** | N | N | N | N | N |
| | | ***strcpy*** | N | Y | N | Y | N |
| | | strcat | N | Y | N | Y | N |
| | | *gets* (usage) | Y | N | N | N | N |
| | | sprintf | N | Y | N | N | N |
| | | strncpy | N | Y | N | Y | N |
| | | strncat | N | Y | N | Y | N |
| | | fgets | N | Y | N | N | N |
| | | snprintf | N | Y | N | N | N |
| ***Format string vulnerabilities*** | | | N | N | N | N | N |
| Memory | ***Double free*** | | Y | Y | Y | Y | Y |
| | ***Improper allocation*** | | N | N | N | Y* | Y* |
| | Initialization errors | ***malloc*** | Y | N | N | Y | Y |
| | | realloc | N | N | N | N | Y |
| | Memory leak | ***malloc*** | Y | Y | Y | Y | N |
| | | calloc | Y | Y | Y | Y | N |
| | | realloc | N | Y | Y | Y | N |
| | ***Failure check*** | | Y | N | Y | N | Y |
| | ***Access freed memory*** | | Y | Y | Y | Y | Y |
| File operations | ***Access closed file*** | | N | N | Y | N | N |
| | ***Access in different mode*** | | N | Y | N | N | N |
| | ***Double close*** | | N | Y | Y | N | N |
| | ***Resource leak*** | | N | Y | Y | N | N |
| | ***Failure check*** | | Y | N | Y | N | N |
| Concurrency errors | ***Deadlock (no multithreading)*** | | N | N | N | N | Y |
| | ***Deadlock (multithreading)*** | | N | N | N | N | N |
| | ***Time Of Check, Time Of Use (TOCTOU) errors*** | | N | N | N | N | N |

| | | Splint | | | Cppcheck | | | Infer | | | Clang | | | Frama-C | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP |
| Path Sensitivity | Simple if-else statement | 89% | 89% | 0% | 83% | 30% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Complex if-else statement | 89% | 100% | 0% | 83% | 30% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Typical for loop | 89% | 100% | 0% | 58% | 100% | 0% | 100% | 0% | 0% | 89% | 0% | 0% | 100% | 0% | 0% |
| | Complex for loop with break command | 100% | 89% | 0% | 58% | 100% | 0% | 89% | 88% | 11% | 89% | 0% | 0% | 100% | 0% | 0% |
| | While loop with continue command | 100% | 89% | 0% | 58% | 100% | 0% | 100% | 0% | 0% | 89% | 0% | 0% | 100% | 0% | 0% |
| | Do-while loop with continue command | 89% | 100% | 0% | 58% | 100% | 0% | 100% | 78% | 0% | 89% | 0% | 0% | 100% | 0% | 0% |
| | Switch statement | 89% | 100% | 0% | 75% | 78% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Goto statement | 89% | 100% | 0% | 58% | 29% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | For loop with arrays | 89% | 100% | 0% | 58% | 100% | 0% | 78% | 100% | 22% | 0% | | 0% | 100% | 0% | 100% |
| | For loop with pointer arithmetic | 100% | 89% | 0% | 58% | 100% | 0% | 67% | 100% | 22% | 0% | | 0% | 100% | 0% | 100% |
| | Conditional operator | 78% | 100% | 0% | 83% | 90% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Return statement | 100% | 78% | 22% | 100% | 25% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Exit function | 89% | 88% | 22% | 83% | 20% | 0% | 89% | 0% | 0% | 89% | 0% | 0% | 100% | 0% | 0% |
| | Define constant | 89% | 100% | 0% | 92% | 18% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Enumeration | 89% | 100% | 0% | 83% | 90% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| Context sensitivity | Simple function calls | 33% | 67% | 56% | 67% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Static variables | 33% | 67% | 22% | 27% | 0% | 0% | 67% | 0% | 0% | 38% | 0% | 0% | 100% | 0% | 0% |
| | Global variables | 22% | 50% | 0% | 27% | 0% | 0% | 67% | 0% | 0% | 38% | 0% | 0% | 100% | 0% | 0% |
| | Function pointers | 44% | 75% | 33% | 0% | | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Structs | 44% | 75% | 33% | 22% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| | Unions | 33% | 67% | 11% | 11% | 0% | 0% | 100% | 0% | 0% | 67% | 0% | 0% | 100% | 0% | 0% |
| | Typedef | 44% | 75% | 33% | 22% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 0% |
| Alias analysis | Direct assignments | 100% | 0% | 0% | 67% | 0% | 0% | 78% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 8% |
| | Casting assignments | 100% | 0% | 0% | 67% | 0% | 0% | 78% | 0% | 0% | 100% | 0% | 0% | 100% | 0% | 8% |

Table 5.2: Performance per type of test of the static analysis tools for *C&K-2011TS*




(a) Performance of the tools on the tests of C&K-2011TS containing a type of defect the tool can detect




(b) Loss of precision when taking all tests from C&K-2011TS into account

Figure 5.1: Recall, precision and $F_{0.5}$ score of the tested tools for C&K-2011TS

kind of test as well as the percentage of tests that yielded one or more unexpected false positives (**uFP**). The tables show areas that are problematic for all of the tested tools.
One of these areas is code containing loops, while the majority of tools (Infer, Clang, and Frama-C) could cope with the simplest of loops consisting of a typical for or while loop, all of the tools struggled with programs in which memory (aside from the iteration counter) is modified and read from within loops. With JM-2018TS memory is modified in all loops leading to a far lower performance in equivalent tests shown in table 5.2 and 5.3. Aside from memory modifications from within loops being problematic, all of the tools struggled with code containing recursive functions. Table 5.2 shows that use of static or global variables led to significantly fewer detections compared to code only containing simple function calls. When no entry point into the test code was provided, as is the case with the results shown in table 5.5, two (Clang and Frama-C) of the tested tools (Infer, Clang, Frama-C, and CppCheck) suffered from a significant loss of precision.

The results show some variation in precision and recall for each kind of test. This variation can in part be explained by the small variations in the structure of the tests, for example, tests for some of the defects may require additional checks to avoid creating additional defects, these checks add a bit of complexity to these tests. The variation in test structure can't explain all of the variation, the remaining variation in results is likely caused by the tools not being equally precise and sensitive for all kinds of defects.

Figures 5.1, 5.2 and 5.3 show what the performance of the tools is for the entire suites in terms of precision, recall, and f-measure. Figure 5.1a shows what the performance of the tools was on the subset of the test suite from *C&K-2011* that contained defects that the tested tool is able to detect. Figure 5.1b shows how precision is affected when false positives from the remaining part of the suite are taken into account. Similarly, figure 5.2a and 5.3a show what the performance of the tools was on the subset of JM-2018TS that contained defects that the tested tool is able to detect and figures 5.2b and 5.3b show how precision is affected when the remaining part of the test suite is taken into account. Here 5.2 shows results for JM-2018TS when an entry point was provided while 5.3 shows results for the situation in which this was not the case.

## 5.3 Performance per tool

So far we have only looked at general trends, we will now look in more detail on how each of the tools performed in both of the test suites.

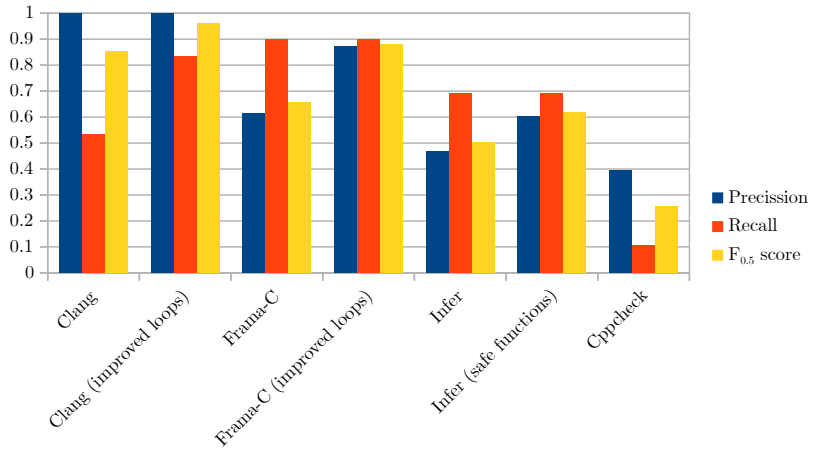
Splint's $F_{0.5}$ score was the lowest of all of the tested tools. In *C&K-2011TS* it had a precision of 54%, which dropped to 51% when taking false positives from the entire test suite into account. In nearly every path-sensitive test a false positive was detected, meaning that the tool has extremely poor path sensitivity. Splint fared a little better precision-wise when looking at context-sensitive tests, yet its performance remained very poor. The recall for the context-sensitive tests was the lowest of all of the tested tools.

| | Clang | | | Frama-C | | | Infer | | |
|---|---|---|---|---|---|---|---|---|---|
| | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP |
| If (boolean) | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 100% | 14% |
| If (integer) | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 100% | 0% |
| If (boolean, multiple functions) | 90% | 0% | 0% | 100% | 0% | 0% | 57% | 100% | 0% |
| Switch | 100% | 0% | 0% | 100% | 0% | 0% | 100% | 100% | 0% |
| Goto | 100% | 0% | 0% | 100% | 100% | 20% | 100% | 100% | 14% |
| Pass by reference | 100% | 0% | 0% | 100% | 10% | 0% | 100% | 100% | 14% |
| Cross file analysis | 0% | | 0% | 100% | 10% | 0% | 100% | 100% | 14% |
| Loop: for | 0% | | 0% | 100% | 100% | 20% | 71% | 100% | 57% |
| Loop: for, complex | 0% | | 0% | 100% | 100% | 20% | 100% | 100% | 57% |
| Loop: while+continue | 0% | | 0% | 100% | 100% | 20% | 71% | 100% | 43% |
| Loop: do-while+continue | 0% | | 0% | 100% | 100% | 10% | 71% | 100% | 43% |
| Loop: for, branching based on values in array | 0% | | 0% | 100% | 100% | 90% | 71% | 100% | 43% |
| Loop: for, pointer arithmetic | 0% | | 0% | 100% | 100% | 100% | 0% | | 0% |
| Loop: recursion | 0% | | 0% | 0% | | 0% | 14% | 100% | 14% |
| Loop: recursion, multiple functions | 0% | | 0% | 0% | | 0% | 29% | 100% | 43% |
| Pseudo recursion | 100% | 0% | 0% | 100% | 0% | 0% | 29% | 100% | 0% |
| Function pointers | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Accessing structs | 90% | 0% | 0% | 100% | 10% | 0% | 86% | 100% | 0% |
| Struct with counter | 90% | 0% | 0% | 100% | 30% | 90% | 86% | 17% | 0% |
| | Clang (improved loops) | | | Frama-C (improved loops) | | | Cppcheck | | |
| | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP |
| If (boolean) | 100% | 0% | 0% | 100% | 0% | 0% | 18% | 100% | 18% |
| If (integer) | 100% | 0% | 0% | 100% | 0% | 0% | 18% | 100% | 18% |
| If (boolean, multiple functions) | 90% | 0% | 0% | 100% | 0% | 0% | 18% | 100% | 27% |
| Switch | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Goto | 100% | 0% | 0% | 100% | 100% | 20% | 0% | | 18% |
| Pass by reference | 100% | 0% | 0% | 100% | 0% | 0% | 9% | 100% | 0% |
| Cross file analysis | 0% | | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: for | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 18% |
| Loop: for, complex | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 18% |
| Loop: while+continue | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: do-while+continue | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: for, branching based on values in array | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: for, pointer arithmetic | 100% | 0% | 0% | 100% | 10% | 0% | 0% | | 0% |
| Loop: recursion | 0% | | 0% | 0% | | 0% | 9% | 0% | 18% |
| Loop: recursion, multiple functions | 0% | | 0% | 0% | | 0% | 9% | 0% | 27% |
| Pseudo recursion | 100% | 0% | 0% | 100% | 0% | 0% | 9% | 100% | 36% |
| Function pointers | 100% | 0% | 0% | 100% | 0% | 0% | 9% | 100% | 0% |
| Accessing structs | 90% | 0% | 0% | 100% | 10% | 0% | 0% | | 9% |
| Struct with counter | 90.00% | 0.00% | 0.00% | 100.00% | 30.00% | 90.00% | 9% | 100% | 18% |

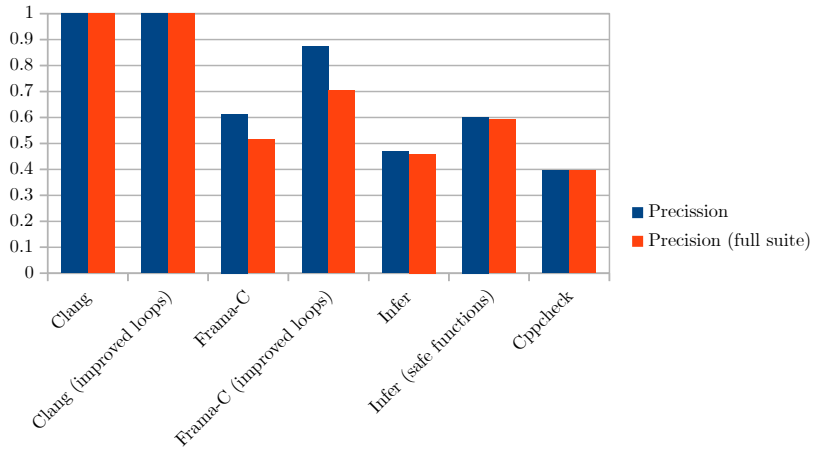
Table 5.3: Analysis sensitivity of the tested tools for tests from *JM-2018TS*

| | Infer (safe functions) | | |
|---|---|---|---|
| | Recall | eFP/TP | uFP |
| If (boolean) | 100% | 29% | 14% |
| If (integer) | 100% | 14% | 0% |
| If (boolean, multiple functions) | 57% | 0% | 0% |
| Switch | 100% | 100% | 0% |
| Goto | 100% | 14% | 14% |
| Pass by reference | 100% | 0% | 14% |
| Cross file analysis | 100% | 29% | 14% |
| Loop: for | 71% | 100% | 57% |
| Loop: for, complex | 100% | 100% | 57% |
| Loop: while+continue | 71% | 100% | 43% |
| Loop: do-while+continue | 71% | 100% | 29% |
| Loop: for, branching based on values in array | 71% | 100% | 43% |
| Loop: for, pointer arithmetic | 0% | | 0% |
| Loop: recursion | 14% | 0% | 0% |
| Loop: recursion, multiple functions | 29% | 0% | 29% |
| Pseudo recursion | 29% | 50% | 0% |
| Function pointers | 0% | | 0% |
| Accessing structs | 86% | 0% | 0% |
| Struct with counter | 85.71% | 16.67% | 0.00% |

Table 5.4: Addition to table 5.3: Infer's performance when the functions in $T_{correct}$ abort when called with dangerous input.




(a) Performance of the tools on the tests of *JM-2018TS* containing a type of defect which the tool can detect

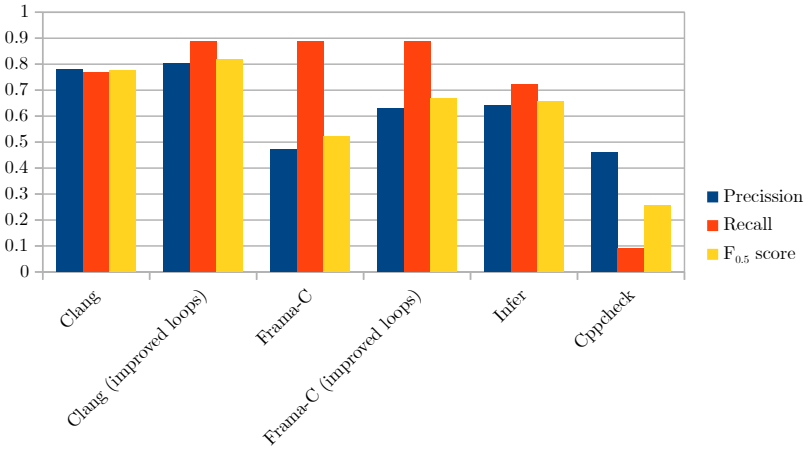



(b) Loss of precision when taking all of the tests from *JM-2018TS* into account

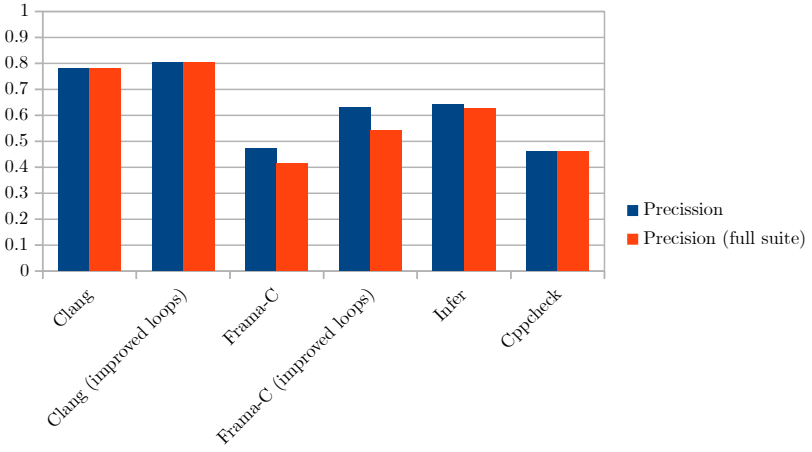
Figure 5.2: Recall, precision and $F_{0.5}$ score of the static analysis tools when testing the new suite. Entry points to the code present in the test cases were provided.

| | Clang | | | Frama-C | | | Infer | | |
|---|---|---|---|---|---|---|---|---|---|
| | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP |
| If (boolean) | 100% | 0% | 0% | 100% | 100% | 0% | 100% | 29% | 14% |
| If (integer) | 100% | 100% | 10% | 100% | 100% | 0% | 100% | 14% | 0% |
| If (boolean, multiple functions) | 90% | 0% | 0% | 100% | 100% | 0% | 57% | 0% | 0% |
| Switch | 100% | 10% | 10% | 100% | 100% | 0% | 100% | 86% | 0% |
| Goto | 100% | 0% | 0% | 100% | 100% | 20% | 100% | 0% | 0% |
| Pass by reference | 100% | 0% | 0% | 100% | 100% | 0% | 100% | 0% | 14% |
| Cross file analysis | 0% | | 0% | 100% | 100% | 0% | 100% | 14% | 14% |
| Loop: for | 90% | 0% | 0% | 100% | 100% | 20% | 71% | 100% | 43% |
| Loop: for, complex | 90% | 0% | 0% | 100% | 100% | 20% | 100% | 100% | 43% |
| Loop: while+continue | 90% | 0% | 0% | 100% | 100% | 20% | 71% | 100% | 14% |
| Loop: do-while+continue | 90% | 0% | 0% | 100% | 100% | 10% | 71% | 100% | 14% |
| Loop: for, branching based on values in array | 0% | | 0% | 100% | 100% | 90% | 71% | 100% | 29% |
| Loop: for, pointer arithmetic | 20% | 0% | 0% | 100% | 100% | 100% | 0% | | 0% |
| Loop: recursion | 70% | 43% | 0% | 0% | | 0% | 14% | 0% | 0% |
| Loop: recursion, multiple functions | 80% | 50% | 0% | 0% | | 0% | 43% | 0% | 0% |
| Pseudo recursion | 70% | 86% | 0% | 100% | 100% | 0% | 14% | 100% | 0% |
| Accessing structs | 90% | 100% | 30% | 100% | 100% | 0% | 86% | 0% | 0% |
| | Clang (improved loops) | | | Frama-C (improved loops) | | | Cppcheck | | |
| | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP | Recall | eFP/TP | uFP |
| If (boolean) | 100% | 0% | 0% | 100% | 100% | 0% | 18% | 100% | 18% |
| If (integer) | 100% | 100% | 10% | 100% | 100% | 0% | 18% | 100% | 18% |
| If (boolean, multiple functions) | 90% | 0% | 0% | 100% | 100% | 0% | 18% | 100% | 27% |
| Switch | 100% | 10% | 10% | 100% | 100% | 0% | 0% | | 0% |
| Goto | 100% | 0% | 0% | 100% | 100% | 20% | 0% | | 18% |
| Pass by reference | 100% | 0% | 0% | 100% | 100% | 0% | 9% | 100% | 0% |
| Cross file analysis | 0% | | 0% | 100% | 100% | 0% | 0% | | 0% |
| Loop: for | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 18% |
| Loop: for, complex | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 18% |
| Loop: while+continue | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: do-while+continue | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: for, branching based on values in array | 100% | 0% | 0% | 100% | 0% | 0% | 0% | | 0% |
| Loop: for, pointer arithmetic | 100% | 0% | 0% | 100% | 20% | 0% | 0% | | 0% |
| Loop: recursion | 70% | 43% | 0% | 0% | | 0% | 0% | | 0% |
| Loop: recursion, multiple functions | 80% | 50% | 0% | 0% | | 0% | 0% | | 0% |
| Pseudo recursion | 70% | 86% | 0% | 100% | 100% | 0% | 0% | | 0% |
| Accessing structs | 90% | 100% | 30% | 100% | 100% | 0% | 0% | | 0% |

Table 5.5: Analysis sensitivity of the tested tools for tests from *JM-2018TS* when no entry point is provided




(a) Performance of the tools on the tests of *JM-2018TS* containing a type of defect which the tool can detect when no entry point into the code is provided




(b) Loss of precision when taking into account all tests from *JM-2018TS* (no entry point provided)

Figure 5.3: Recall, precision and $F_{0.5}$ score of the static analysis tools when testing the *JM-2018TS* for which no entry point was provided for the code present in the test cases

One part of the test suite where this tool did perform well was on the alias analysis tests, here it did not miss any defects, while not yielding any false positives.

Cppcheck had the second lowest $F_{0.5}$ score for *C&K-2011TS* and the lowest $F_{0.5}$ scores for *JM-2018TS*. For both test suites, it had the lowest recall of all tools. Both the recall and precision of Cppcheck were far lower for *JM-2018TS* than what was measured for *C&K-2011TS*.

For every type of test Infer's recall and precision were as good or better than those of Cppcheck, yet overall it was outperformed by both Clang and Frama-C. Infer had the third highest $F_{0.5}$ scores for *C&K-2011TS* and *JM-2018TS,* with an exception being *JM-2018TS* in the case where no entry point into the source code is provided. In this case, it achieved the second highest $F_{0.5}$ score if the default configuration was used for all tools.

Frama-C performed very well for both *C&K-2011TS* and *JM-2018TS* in the case where an entry point into the code was provided, achieving the second highest $F_{0.5}$ score for both test suites. Frama-C performed significantly worse for *JM-2018TS* in the case where no entry point into the test functions is provided. Here Frama-C was overtaken by Infer when looking at the $F_{0.5}$ score when the default configuration was used. As the only correct tool in the selection of tools tested Frama-C was the only tool that was able to detect all defects of the type it is capable of finding. (Recursion was explicitly not supported by this tool.)

In all situations, Clang had the highest $F_{0.5}$ score. Clang had the highest precision of all tools while its recall was generally trailing behind Frama-C and Infer.

## 5.4 Timed real world example

| Analysis tool | Analysis time |
|---|---|
| Cppcheck (1 thread) | 106 minutes |
| Cppcheck (4 threads) | 145 minutes |
| Infer | 51 minutes |
| Clang | 22 minutes |

Table 5.6: Time spent analyzing Gimp 2.8 code

To give some insight into how the tools perform execution time-wise in a complex real-life example Gimp version 2.8.22 was analyzed by all tools[2]. This application was chosen as it is a well known open source application, the source of which contains 774554 lines of code in 2975 C files[3] (`.c`, `.h`). The time it took each tool to complete analysis is shown in table 5.6. Both Splint and Frama-C failed to analyze this application. Splint

[2]The analysis was run on a dual-core Intel Core i7 4500u processor with 9.2 GiB of accessible ram.
[3]Numbers acquired with the `cloc` utility

was terminated after 36 minutes had elapsed while its memory usage had reached 4.5 GiB and was still increasing while it showed no sign of progress. Frama-C aborted with a syntax error in one of the source files stated as the reason. Cppcheck performed significantly better when 1 thread was used, which is the default, instead of 4 threads. The most prevalent kinds of defects detected by the static analysis tools can be seen in table 5.7, problems that do not impact security or reliability have been excluded from the results. These detections have not been verified, what percentage of detections are valid is thus unknown.

| | Cppcheck | Infer | Clang |
|---|---|---|---|
| Null pointer | 15 | 198 | 220 |
| Memory leak | 0 | 22 | 62 |
| Resource leak | 16 | 16 | 0 |
| Use after free | 0 | 13 | 3 |
| Uninitialized variable | 83 | 0 | 22 |
| Defects found: Total | 130 | 254 | 449 |

Table 5.7: Most prevalent kinds of defects (possibly FP) found in Gimp 2.8 code

## 5.5 Usability

### 5.5.1 Splint

Splint provided some basic information about the detected defects: line number of the offending statement, the name of the variable used (if applicable) and a short description of the kind of defect detected. If the statement is incorrect only because of the current state of the memory used in that statement the line number in which this memory is last changed is also provided.

When using the default configuration, Splint's output contained a large amount of noise. As this program is a linter it is to be expected that not all output is related to a defect. Unfortunately, part of the security-related warnings is merely about the usage of certain constructs and not about whether this usage was not correct.

Splint offers no options to sort the output by defect type or severity. It is possible to filter out certain warnings, by supplying command line flags to suppress these warnings. Splint does not support generating its output in a common file format like XML or JSON, which could have facilitated filtering and sorting the output externally.

The only way to supply Splint with information on which files should be analyzed and which libraries to include is to supply these libraries and files as command line parameters. This results in a poor user experience when working on complex projects.

Listing 5.1: Example output from Splint

```
1 memory_doublefree1_alias1.c: (in function main)
2 memory_doublefree1_alias1.c:16:10: Variable c used after being released
3 Memory is used after it has been released (either by passing as an only
    ↪ param or assigning to an only global). (Use -usereleased to inhibit
    ↪  warning)
4 memory_doublefree1_alias1.c:14:10: Storage c released
5
6 Finished checking --- 1 code warning
```

### 5.5.2 Cppcheck

Cppcheck provides roughly the same information as Splint, except for that it doesn't provide information on prior modifications of memory and the description of the defect is more basic. Even though it does offer a little less information than Splint, it makes up for this by warning about a far more useful selection of issues: all warnings were related to defects and of these warnings, all were referring to incorrect usage of a construct instead of merely the usage itself.

Cppcheck does not offer any options to sort its output by severity and defect type. Cppcheck's ability to filter out certain types of defects is limited to suppressing warnings supplied as command line flags. Cppcheck can output its result in both XML and PLIST format, which may simplify using its output in a program capable of sorting and filtering this output.

Cppcheck does support analyzing an entire code base by supplying it with a compilation database. This can make using the tool on existing code significantly easier as it removes the need to manually supply the paths to all source files. Using a compilation database removes the need to manually supply information on which libraries need to be included. Finally, a compilation database contains information on which preprocessor directives should be used for building the application or library ensuring that the combination of directives tested is a valid one. Cppcheck is the only tool of the ones tested in this article capable of analyzing code with missing dependencies (analyzing part of the source code, missing header inclusions).

Listing 5.2: Example output from Cppcheck

```
1 [memory_doublefree1_alias1.c:19]: (error) Memory pointed to by 'c' is
    ↪ freed twice.
2 [memory_doublefree1_alias1.c:19]: (error) Deallocating a deallocated
    ↪ pointer: c
```

### 5.5.3 Infer

Infer provides a short description of the defect including the variable used, the function called and the line at which the variable used is last modified if this is relevant. Infer's output includes a snippet of code containing the problematic statement, which provides some context and could make it easier for the user to find the offending statement in the code. Infer outputs some files that include, among other things, some information about the path taken for each defect. The assumptions made when this path was taken are however not available.

Like Cppcheck Infer is able to use a compilation database to analyze an application or library. Additionally, it can learn this information by supplying Infer with the build commands as it is able to analyze the compilation of the code for a wide selection of build systems.

Listing 5.3: Example output from Infer

```
1  memory_doublefree1_alias1.c:19: error: USE_AFTER_FREE
2  pointer `c2` last assigned on line 17 was freed by call to `free()` at
       ↪ line 14, column 5 and is dereferenced or freed at line 19, column 5
3    17.         c2 = c1;
4    18.
5    19. >       free(c2); /* DANGER */
6    20.
7    21.         return 1;
8
9  Summary of the reports
10
11   USE_AFTER_FREE: 1
```

### 5.5.4 Clang

Clang's command line output is limited to a basic description of the kind of defect, line number of the offending statement and the offending statement itself. By default, the scan-build utility also generates HTML output for each defect. In this file, the offending code is annotated with both the path that was taken as well as the assumptions that were made in reaching the offending statement. As a statement may not lead to a defect for all paths in which it can be reached nor for all possible variable values, this information could help the user understand in what conditions the defect arises. An example of source code annotated by Clang can be seen in figure 5.4.

In the overview present in the HTML output the user can sort the defects by type or filename and filter out types of defects.

The scan-build utility used does not support using a compilation database, but, like Infer, it can get this information by building the code when the build commands are supplied to the tool.

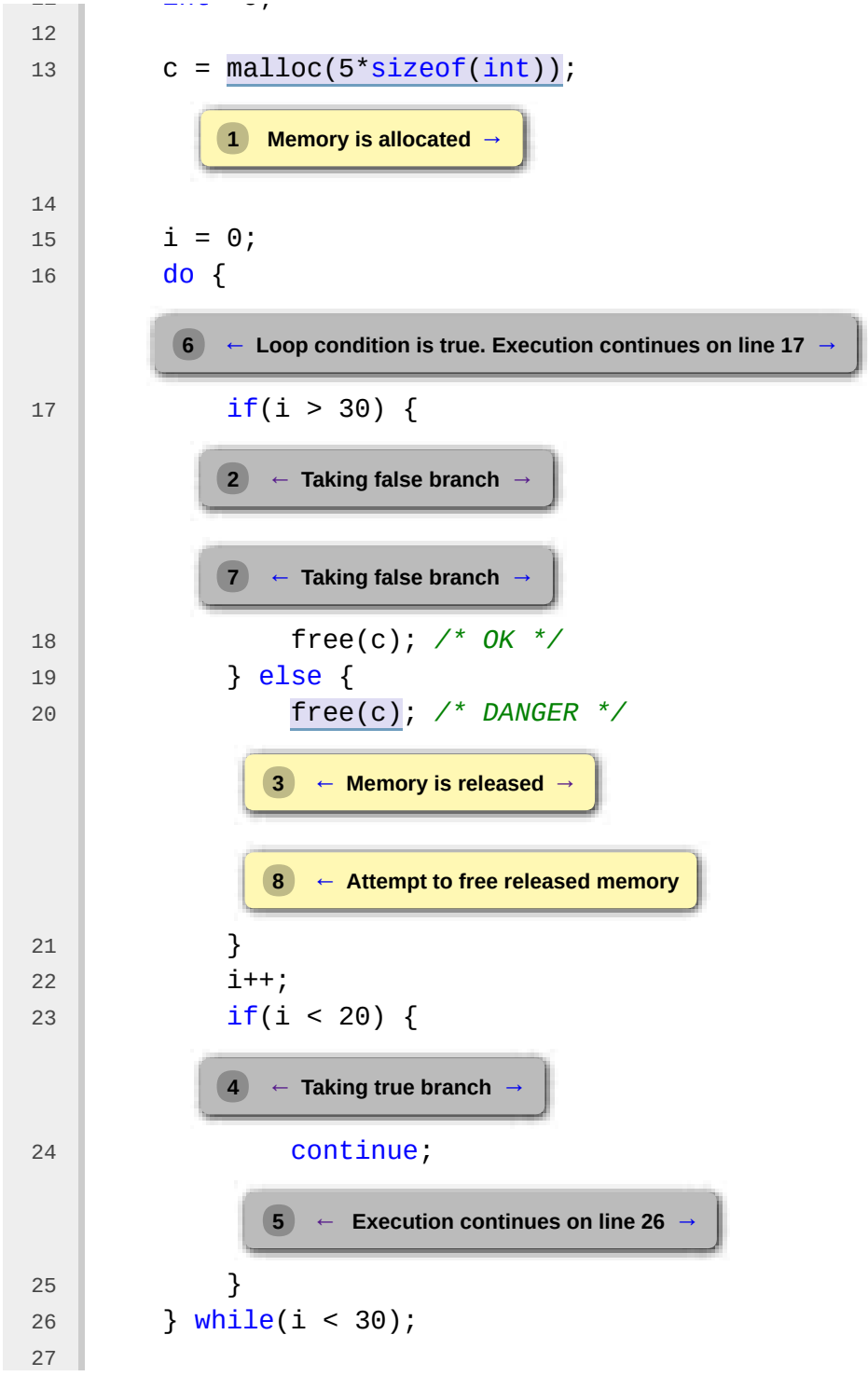



Figure 5.4: Example output from Clang

Listing 5.4: Example output from the scan-build frontend for Clang

```
1 scan-build: Using '/usr/lib/llvm-5.0/bin/clang' for static analysis
2 memory_doublefree1_alias1.c:19:5: warning: Attempt to free released
    ↪ memory
3     free(c2); /* DANGER */
4     ^~~~~~~~
5 1 warning generated.
6 scan-build: 1 bug found.
7 scan-build: Run 'scan-view /tmp/scan-build-<directory suffix> to examine
    ↪ bug reports.
```

### 5.5.5 Frama-C

Frama-C's command line output was very verbose, most information given is not relevant to any detected defects. The detected defects are described by giving a technical description of the kind of defect and the assertion that failed where the other tools' descriptions are closer to normal sentences. The information about the issue is limited to line-number and the failed assertion which contains the variable related to the defect. Compared to the other tools it offers some unique information: the domains of the final values of the variables present in the analyzed functions. With the GUI one can view the value domain of a variable at any point in the code. Using the GUI the location of the defects can be reached by selecting the warning.

Frama-C does not support supplying compilation information by supplying a compilation database nor is it able to acquire this information by analyzing the build process.

Listing 5.5: Example output from Frama-C

```
[kernel] Parsing memory_doublefree1_alias1.c (with preprocessing)
[value] Analyzing a complete application starting at main
[value] Computing initial state
[value] Initial state computed
[value:initial-state] Values of globals at initialization
  __fc_random_counter ∈ {0}
  __fc_rand_max ∈ {32767}
  __fc_heap_status ∈ [--..--]
  __fc_mblen_state ∈ {0}
  __fc_mbtowc_state ∈ {0}
  __fc_wctomb_state ∈ {0}
memory_doublefree1_alias1.c:12:[value] allocating variable
    ↪ __malloc_main_l12
memory_doublefree1_alias1.c:16:[value] warning: accessing left-value that
    ↪  contains escaping addresses.
                assert ¬\dangling(&c);
[value] done for function main
[value] ====== VALUES COMPUTED ======
[value:final-states] Values at end of function main:
  __fc_heap_status ∈ [--..--]
  c ∈ {{ NULL ; &__malloc_main_l12[0] }} or ESCAPINGADDR
  c1 ∈ {{ NULL ; &__malloc_main_l12[0] }} or ESCAPINGADDR
  c2 ∈ {{ NULL ; &__malloc_main_l12[0] }} or ESCAPINGADDR
  __retres ∈ {1}
```

# Chapter 6

# Discussion

## 6.1 Results

We will look at general properties exhibited by the tested static analysis tools. Appendix A contains the reasoning behind these observations.

Splint's path analysis was the weakest of all tested tools which caused this tool to achieve very low precision in nearly all tests.

Cppcheck's path sensitivity proved to be sufficient only in simple cases while its analysis in situations requiring alias analysis was the least-sensitive of all tested tools. It is generally unable to find defects in code in which a fault occurs for some, but not all of the possible execution paths. Additionally, it is not as precise as the other tools (barring splint) when it does find issues. It may be unable to detect that code is unreachable which may lead to false positives in this code. Cppcheck struggled to detect issues consisting of incorrect usage of memory that can not directly be accessed from a variable (e.g. global variables, struct members, etc.) in the current function.

Infer performed well in most cases, generally achieving a high precision and recall. Infer did however not achieve good precision in tests containing loops and offered no options to improve this precision. A potential issue that was found is that Infer analyzes for parameters that will not occur during the runtime of the program analyzed. Infer did generally perform better than any other tool in cases in which variables are compared to other variables of which the exact value is unknown. However, in cases where the values of the memory used can be fully determined it generally did not perform as well as Clang or Frama-C.

Clang's precision was higher than any other tool tested. Its abstract domain did however not seem expressive enough to represent numeric states relative to other numeric variables, which led to low precision in cases where two integers of unknown value are compared. Another issue was that Clang was not able to do cross-file analysis which will limit its recall in practice as dangerous calls to functions defined in other `.c` files will

not be detected. Clang's loop analysis was excellent as long as the number of iterations per loop to analyze does not exceed the configured maximum prior to encountering a defect, otherwise, no defects were detected.

Frama-C performed well in sequential cases in which the exact value of all memory can be known. In cases where code branches based on a comparison that includes a variable of which the exact value is not known it achieved very low precision. This is caused by its abstract domain not being expressive enough for the type of analysis used. When Frama-C's default configuration was used Frama-C was not able to achieve good precision for most tests containing loops. Frama-C can, however, be configured to achieve better precision in these cases.

## 6.2 Usability

Frama-C is not suitable for use as a general purpose static analysis tool because of the verbosity of its output and the amount of manual work needed. Using Frama-C requires a significant amount of manual work in order to analyze an entire application. Analyzing a library requires additional work as this can only be done by analyzing a single exported function at a time. The verbosity of the output renders it unsuitable to analyze large amounts of code on a frequent basis. When used to analyze smaller, self-contained pieces of code near the end of development these issues have less of a negative impact. In that case, Frama-C's high recall may be worth the larger amount of effort required to gain and analyze results.

Infer and Clang are both very suitable for analyzing complex applications as both integrate well with a large selection of build systems and output clear descriptions of issues. Clangs ability to output the which path was taken and assumptions were made can be especially useful when looking at issues in complex code.

Even though Cppcheck's path-analysis is too simple to perform well on complex code it does have some advantages over the other tools. It can detect a wide array of defects and secondly, it is able to analyze incomplete code. This last property can be especially useful as it allows for using this tool on code that is still being written.

## 6.3 Related work

This article is heavily influenced by Chatzieleftheriou and Katsaros [2011] and this article originally included an attempt to reproduce the results from this original article. This attempt failed as the measured recall for both Splint and Cppcheck did not match the recall described in the original article[1]. I have described the method used to acquire recall

[1]The measured recall was higher than expected: a ˜15 percent point difference with what was found in [Chatzieleftheriou and Katsaros, 2011]. The difference in measured precision was within 2 percent point.

and precision in more detail and hope that this will lead to more success in reproducing the results from this article, should one want to do so. The kind of defects the tools were capable of finding did also not match the original article[2]. Chatzieleftheriou and Katsaros [2011] includes a quantitative test suite to measure how the memory usage and execution time of tools scales with increasingly large source files. These tests consist of repeating segments of code making it trivial to generate larger source files while no other conditions are changed. An increasing number of source files as well as an increase of dependencies between pieces of code, what one would expect for a growing code base, is unfortunately not taken into account. Measuring the effects of increasing code size on execution time and memory scaling using real code bases alongside these tests may give a more complete picture. Measuring execution time using several programs of varying complexity may be an interesting subject for further research.

In McLean [2012], Mantere et al. [2009] the usability of tested tools is described in more detail. Both of these articles test a small number of static analysis tools (McLean benchmarks two tools while the article from Mantere et al. benchmarks three tools). These articles are less suitable for determining how well a tool performs as the testing method does not generate conclusive results. McLean [2012] is less suitable as the output of the tools is not checked for false positives. The test suite used in Mantere et al. contains 18 test cases total for 5 different kinds of defects. This number is too small to test the tools in a manner similar to this article.

Newsham and Chess [2006], Liu et al. [2012] propose an automatic benchmark system in which the performance of static analysis tools can be measured automatically. The way these test suites benchmark the tools is different from the test suites used in this article. The test suites do not seem to systematically test the recall and precision for each kind of defect in different situations.

Zitser et al. [2004] measured the ability of static analysis tools to find real buffer overflow defects by testing these tools on versions of several open source programs that contained exploitable buffer overflows, making its scope more limited and the used method different.

[2]According to my findings Splint is not able to detect incorrect usage of *sprintf* , but instead warns that sprintf is used. Similarly, it was not able to detect *sign errors*, it did instead warn about any usage of signed integers in an unsigned integer context. All tools that are capable of finding the usage of uninitialized pointers were found to able to find the usage of unopened files (usage of uninitialized file pointers), while according to the original article none of the tools were able to find this kind of defect.

# Chapter 7

# Conclusion

We have looked at in what situations the tested tools perform well in terms of precision and recall and have evaluated the usability of several open source static analysis tools. To accomplish this a new test suite addressing one of the issues of *C&K-2011TS* was used. This new test suite also increased the number of scenarios in which a defect can occur. Using this new suite allows us to observe shortcomings of static analysis tools that could not be detected when *C&K-2011TS* was used on its own. Most notably it allowed us to see what the performance of static analysis tools is in cases where the value of function parameters is not known and/or functions are defined in separate C file. This test suite is freely available at `https://github.com/JMoerman/JM2018TS`. The results show that both Infer and Clang can compete with Frama-C, which in previous articles often performed significantly better than other open source static analysis tools. Clang's ability to annotate source code with information on what circumstances may lead to a fault was also found to be useful as this may ease the analysis of the detected issues.

# Chapter 8

# Bibliography


Clang static analyzer project website. `https://clang-analyzer.llvm.org/` - accessed 5-February-2018.

Cppcheck project website. `http://cppcheck.sourceforge.net/` - accessed 3-February-2018.

Frama-c project website. `https://frama-c.com` - accessed 4-February-2018.

Infer website. `http://fbinfer.com` - accessed 4-February-2018.

Clang static analyzer (scan-build utility) project website.

Splint project site on virginia.edu. `http://lclint.cs.virginia.edu/` - accessed 3-February-2018, mirror of inaccessible `http://splint.org` site.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for c programs. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 123–124. IEEE, 2009.

George Chatzieleftheriou and Panagiotis Katsaros. Test-driving static analysis tools in search of c code vulnerabilities. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 96–103. IEEE, 2011.

Nancy Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th conference on Message understanding*, pages 22–29. Association for Computational Linguistics, 1992.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

Jiangchao Liu, Liqian Chen, Longming Dong, and Ji Wang. Ucbench: A user-centric benchmark suite for c code static analyzers. In *Information Science and Technology (ICIST), 2012 International Conference on*, pages 230–237. IEEE, 2012.

Matti Mantere, Ilkka Uusitalo, and Juha Roning. Comparison of static code analysis tools. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*, pages 15–22. IEEE, 2009.

Daniel Marjamäki. Cppcheck design, 2010. `http://sourceforge.net/projects/cppcheck/files/Articles/cppcheck-design-2010.pdf`.

Ryan K McLean. Comparing static security analysis tools using open source software. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 68–74. IEEE, 2012.

Tim Newsham and Brian Chess. Abm: A prototype for benchmarking source code analyzers. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics. US National Institute of Standards and Technology (NIST) Special Publication (SP)*, pages 500–265, 2006.

Lucas Torri, Guilherme Fachini, Leonardo Steinfeld, Vesmar Camara, Luigi Carro, and Érika Cota. An evaluation of free/open source static analysis tools applied to embedded software. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1–6. IEEE, 2010.

Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

# Appendix A

# Results in detail

## A.1 Splint

In *C&K-2011TS* Splint achieved the lowest precision of all of the tools: a precision of 54%, which dropped to 51% when taking false positives from the entire test suite into account. In nearly every path-sensitive test a false positive was detected, meaning that the tool has extremely poor path-sensitivity. Splint fared a little better precision wise when looking at context-sensitive tests, yet it remained very poor. The recall for the context-sensitive tests was the lowest of all of the tested tools. One part of the test suite where this tool did perform well was on the alias analysis tests, here it did not miss any defects, while not yielding any false positives.

## A.2 Cppcheck

In *C&K-2011TS* Cppcheck's performance was held back mainly by lower precision in the path-sensitive tests and generally low recall. The false positives in Cppcheck's output consist of detections of possible problems in statements that cannot be reached. When we look at Daniel Marjamäki's article on Cppcheck design [Marjamäki, 2010], a good explanation for these false positives can be found. According to this article, Cppcheck assumes that all statements are reachable. If it could be assumed that these statements were indeed reachable then these statements would indeed have caused a fault where the statement to be executed after the last reachable previous statement. Cppcheck's recall for the alias analysis tests was the lowest of all tools. Cppcheck had a low recall for tests which depend on accessing memory (global variables, struct members, etc.) not directly accessible from variables defined in the function being analyzed. Cppcheck was the only tool not capable of finding defects in tests contain functions reached by a function pointer.

Cppcheck performed significantly worse in *JM-2018TS*. Here the tool detected nearly no defects while precision, for tests containing if statements, is reduced to 50% where this

was ˜77% for *C&K-2011TS*. A difference between the two test suites is that all of the test programs in *JM-2018TS* consist of multiple functions. The usage of function calls can result in a lower recall as can be seen from the *Context sensitivity: simple function calls results* in table 5.2. However, even when rewriting tests containing if statements to only contain a single function the low recall remained. This means that the difference in results can instead be attributed to the difference in the structure of the tests. Unlike the path-sensitive tests in *C&K-2011TS,* for most tests in *JM-2018TS,* all statements can be reached without a fault occurring. Whether a statement will cause a fault in the majority of tests from *JM-2018TS* is determined by which path was taken prior to reaching a statement. This means that the path by which a statement can be reached must be taken into account[1], something Cppcheck struggles with.

## A.3 Frama-C

Frama-C performed very well in most cases, however, non-sequential code and code where the exact code of parameters is not known was problematic.

Frama-C does not consider each path separately but instead unifies paths by unifying the possible states of each piece of memory for each path. An example of this can be seen in figure A.1 showing how Frama-C joins two paths that split based on an unknown value of $A$. Reducing the number of paths to analyze in this way creates problems, however, as the set of possible states of the entirety of the program memory can then include states that are not possible during execution of the code. This issue causes Frama-C to emit false positives mainly in situations where the exact value of the memory in use cannot be determined, Listing A.1 provides an example of code for which this issue causes Frama-C to emit a false positive. This issue caused Frama-C to perform very poorly on all tests that do not belong the *loop* category for *JM-2018TS* in the case where the value of the parameters is not known.

Frama-C also struggles with memory mutations within loops when loops are not fully unrolled. The new range of possible states for a piece of memory that results from an assignment to this memory can be larger than is possible during execution. This issue was the cause of the false positives in tests containing `for` or `while` loops for tests from both *JM-2018TS* and *C&K-2011.* An example of this issue resulting in false positives can be seen in listing A.2. Fortunately, Frama-C includes an option to unroll loops to a configurable amount. By configuring Frama-C to unroll sufficient iterations of each loop good precision could be achieved for these tests.

[1] In *C&K-2011TS* all that needs to be considered is if a statement can be reached, not how.

Listing A.1: Example of correct code for which Frama-C detects an issue as dependencies between values are not taken into account

```
1  void safe_function() {
2      int* data;
3      bool a = rand() & 1;
4      bool b = rand() & 1;
5
6      if(!a && !b) {return;}
7
8      data = malloc(20 * sizeof(int));
9      ... // Code contains check causing the function to terminate if 'data
           ↪ ' is NULL.
10 // Pointer 'data' points to the memory data at line 8.
11
12     if(a) {
13         free(data);
14 // Frama-C: Pointer 'data' contains an escaping address.
15     }
16 // Frama-C: At this point 'data' either points to the memory data at line
       ↪  8, or it contains an escaping address. This is correct but
       ↪ imprecise as we know it will only contain an escaping address if a
       ↪ was true.
17
18     ...
19
20     if(b && !a) {
21         free(data);
22 // Frama-C warns that it is possible that 'data' is accessed while it
       ↪ contains an escaping address, which is incorrect as line 13 can not
       ↪  be reached if ¬a
23     }
24
25 // Final state of 'data' according to Frama-C: 'data' can be NULL, can
       ↪ point to the memory data at line 8, can be uninitialized or
       ↪ contains an escaping address. In reality, it can not point to the
       ↪ memory data at line 8 as ¬(¬a ∧ ¬b) ∧ ¬a ∧ ¬(b ∧ ¬a) can not be true
26 }
```

Listing A.2: Example of imprecise loop analysis by Frama-C

```
1  int arr[10];
2  int i;
3  int a = 0;
4
5  for(i = 0; i < 10; i++) {
6      arr[i] = i;
7  }
8
9  // According to Frama-C the elements in arr will at this point be
       ↪ initialized to 0-9 or are uninitialized. In reality arr will always
       ↪  be fully initialized as the loop above is guaranteed to initialize
       ↪  the entire array.
10
11 for(i = 0; i < 10; i++) {
12     a += arr[i];
13 // Frama-C warns here that it is possible for arr[i] to not be
       ↪ initialized. Frama-C also warns of a possible integer sign overflow
       ↪ , which is not possible with the values stored in arr. If one or
       ↪ multiple elements of arr were not initialized this would indeed be
       ↪ possible as these elements could have any value.
14 }
15
16 printf("%i\n", a);
17 // According to Frama-C this statement will output a number in the range
       ↪ [0..2147483646] while the only output possible is 45.
```

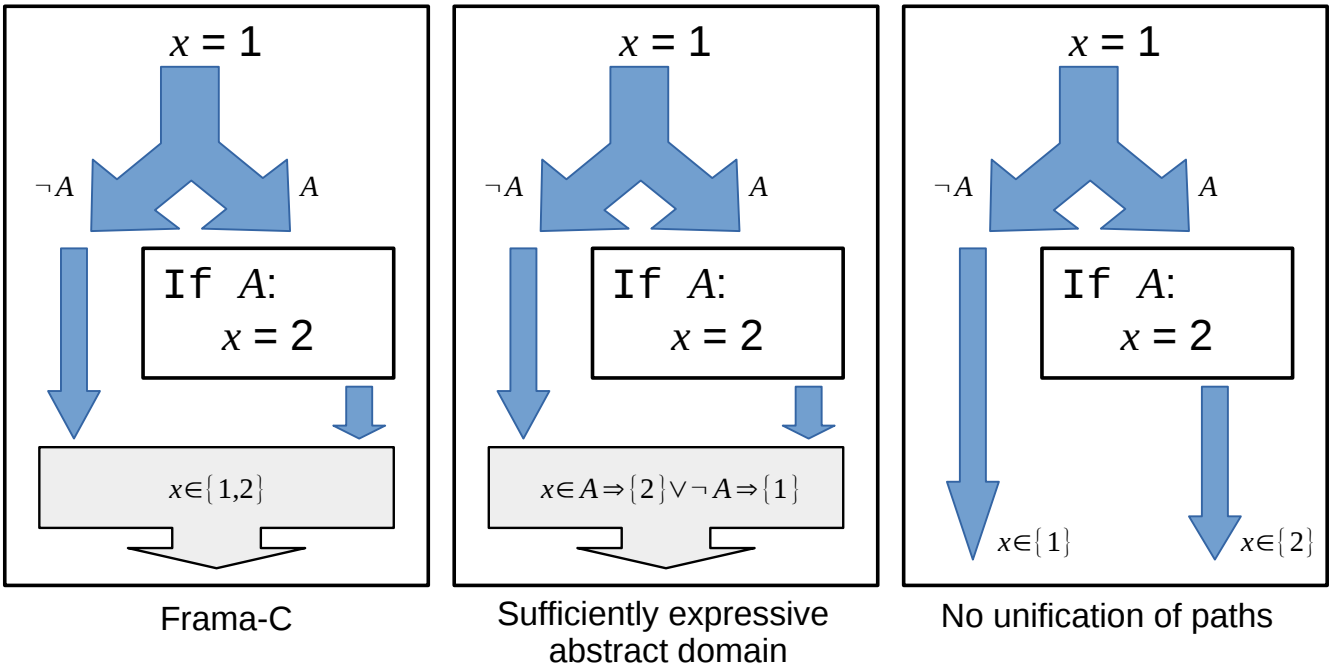



Figure A.1: Figure showing how Frama-C unifies paths as well as two examples that would yield better results

## A.4 Clang

Similar to Frama-C Clang did not perform well on tests where memory modified within loops. When using the default configuration, If memory was modified within a loop containing more iterations ($> 3$) than Clang can analyze no positive of any kind was returned regarding usage of that memory. Like Frama-C Clang offers an option to improve its performance on loops, in this case, the number of times a block of code can be visited can be increased. When increasing this number to n+1, where n is the largest number of iterations present in the code to analyze, Clang found most of the defects while the number of false positives was not increased.

Clang does not support cross-file analysis, when memory was passed to a function located in another `.c` file no positive of any kind was returned regarding usage of that memory.

Clang's abstract domain does not seem to be expressive enough to describe the value of a piece of numerical memory relative to other memory. This was concluded as it is unable to detect which branch can be taken if two integers are compared of which the exact value is not knowable, but of which the ordinal relation can be known. An example of this causing Clang to emit a false positive can be seen in listing A.3. This behavior caused the tool to achieve very low precision for a number of tests from *JM-2018TS* and the source of all of the false positives for this test suite could be traced back to this issue.

Listing A.3: Example of clang detecting an issue in correct code

```
1  void safe_function(int x, int y) {
2      int* data = malloc(sizeof(int));
3
4      if(!data) {
5          return;
6      }
7
8      ...
9
10     if(x <= y) { // Clang: taking true branch
11         free(data); // Clang: Memory is released
12     }
13
14     ...
15
16     if(x > y) { // Clang: taking true branch
17         free(data); // Clang: Attempt to free released memory
18         // Taking the true branch for this if statement and the previous
               ↪ if statement is not possible during execution as the
               ↪ conditions contradict eachother
19     }
20 }
```

Clang's recall for code in which static or global variables are used was far lower than both Infer and Frama-C, tools that have similar performance.

When analyzing the suite from *C&K-2011* it returned a few false positives caused by different causes: poor alias analysis[2] caused two false positives while incorrect modeling of the `realloc` function[3] was responsible for the third false positive.

## A.5 Infer

Like Clang and Frama-C Infer struggled with loops. Unlike these two tools Infer also struggled with some of the tests containing loops from *C&K-2011TS* where memory modifications within the loop were not relevant in code that follows this loop. It also differed from these tools in that it did not provide any means to improve its performance for code containing loops.

Infer performed very poorly when looking at the results of *JM-2018TS* in the case where the entry point is known to the static analysis tool. When the tool was benchmarked in

[2]Clang erroneously detected out-of-bound array access errors when argument A in the function call strncpy(A, B, sizeof(A)) is an alias of a string (Strings, truncation: alias analysis)

[3]Clang also issued a warning (double free) when freeing memory of which reallocation had failed, it seems to assume that realloc frees memory regardless of whether memory could be reallocated successfully.

the same way as all of the other tools the ratio between valid detections and expected false positives was 1:1 for all tests save the *struct with counter* test, which was the only test not relying on parameter values which determine whether a defect is reached or not. When this tool was benchmarked with tests where functions abort when dangerous parameters are supplied the performance, however, greatly increased. Meaning that the lowered precision can largely be attributed to the tool analyzing functions for parameters other than those possible during the execution of the program.

Unlike Clang and Frama-C Infer did not perform worse when no entry point was provided. In this case, Infers performance was comparable or better compared to these tools when tests containing loops are not taken into account.

This tool did have the second lowest performance when looking at the alias analysis tests.

Aside from these issues, Infer did not perform as well as Clang or Frama-C for a number of tests. Unlike Clang and Frama-C Infer was not able to detect defects in tests where a function pointer is passed to a function where it is then called. However, it was able to detect defects in code in which a function pointer is assigned and called within the same function as is the case in the *Context sensitivity: function pointers* tests from *C&K-2011TS*. Infer was also the only tool that was greatly affected by splitting the main function from the *If (boolean)* test into multiple functions. For *JM-2018TS* Infer was not able to distinguish real defects from the expected false positives in the *Switch* test. This test differs from the equivalent test in the other suite in *C&K-2011TS* that it contains an additional if statement which influences the state.

# Appendix B

# Test suite information

## B.1 Types of tests in *JM2018TS*

Most tests consist of several parts

$Mem_{test}$ Condition $C_a$, Condition $C_b$ these conditions can consist of the evaluation of boolean values passed to the test function, or comparison between two integers.

Most tests fall in one of two categories: simple sequential tests and tests containing iterative loops.

### B.1.1 Simple sequential tests

The simple sequential tests consist of the following parts:

1. Program entry point containing a call to the test function.
2. Preparation of the test, variables are initialized to their initial state.
3. De state of $Mem_{test}$ is set, with its value depending on $C_a$. We will call these states $S_{C_a}$ and $S_{\neg C_a}$.
4. Depending on whether condition $C_b$ is true two different things can happen. If $C_b$ is true $Mem_{test}$ is used in a way that is safe regardless of whether $C_a$ was true or false, in some cases $Mem_{test}$ is not used at all. If $\neg C_b$ holds then $Mem_{test}$ is used in a way that is safe only if the state of $Mem_{test}$ is $S_{C_a}$.
5. Finalization of the test: allocated resources not yet freed are freed.

Description of tests within this category:

- **If (boolean):** A simple test where steps 2 up to and including 5 are executed within a single function. This test function is called with two boolean values $a$ and $b$ as parameters. The conditions consist of the evaluation of the boolean values passed to the test function ($C_a = a$, $C_b = b$).

- **If (integer):** A simple test where steps 2 up to and including 5 are executed within a single function. This test function is called with two integer values $x$ and $y$ as parameters. The conditions consist of the comparison of integer values, one comparison between the two parameters and one comparison between one of the parameters and a static value (for example $C_a = (x > 10)$, $C_b = (x \geq y)$).

- **If (boolean, multiple functions):** This test is a variation on the *If (boolean)* test, it differs from this test in that steps 3 and 4 are executed in separate functions called by the main test function containing steps 2 and 5. (The value to assign to $Mem_{test}$ in step 3 is returned by a separate function.)

- **Switch:** This test is a variation on the *If (integer)* test, it differs in that step 4 does not contain a branch based on a single condition, instead this step is built using a switch statement comparing the value of one of the parameters to several static values, some of the values lead to safe usage of $Mem_{test}$ regardless of the outcome of step 3, others lead to usage of $Mem_{test}$ in a way that is unsafe if $C_a$ is false.

- **Pass by reference:** This test is a variation on the *If (boolean)* test, in this test a reference to $Mem_{test}$ is passed to a function which executes step 3.

- **Cross-file analysis:** This test is a variation on the *If (boolean)* test, in this test the value assigned to $Mem_{test}$ in step 3 is generated by a function defined in a c file different from the c file containing the rest of the code of the test.

- **Pseudo recursion:** This test is a variation on the *If (integer)* test with the test function split into two functions $f_1$ and $f_2$. Function $f1$ will call $f2$ if this $f1$ itself wasn't called from $f2$ and vice versa. This test was included as for most tools the recall for tests with recursive functions was far lower than average, this test may help give insight in why this is the case: are functions that appear to be recursive also an issue?

- **Function pointers:** This test is a variation on the *Pass by reference test* with the function executing step 3 passed to the test function instead of being defined there.

- **Accessing structs:** This test is a variation on the *If (integer)* test, it differs from this test in that $Mem_{test}$ is stored in a statically allocated struct instead of being directly accessible via a variable. This struct is accessed via get and set functions.

### B.1.2 Loops

The test functions in these tests take a parameter $x$. Aside from $x$ two variables are used: 1. integer $i$ of which the value is equal to the current iteration of the loop, starting from iteration 0, and test memory $Mem_{test}$.

#### B.1.2.1 Simple Loops

$Mem_{test}$is initialized to a value which is unsafe within the context of all statements within the loop. Each test contains a single loop of 20 iterations. If and only if the loop counter $i$ is larger than parameter $x$ and the memory is in an unsafe state a defect is reached. At the start of the 11th iteration, the memory $Mem_{test}$ is set to a safe state. The initialization of variables and the body of the loop is the same for all of the tests that belong to this category.

- **Loop: for:** This test is built around a typical `for` loop ( `for ( i = 0; i < 20; i ++) {...}`).
- **Loop: for, complex:** This test is built around a typical `for` loop ( `for (;;) {...; i ++; if ( i >= 20) { break ;}}`).
- **Loop: while+continue:** This test is built around a `while` loop one `continue` statement which causes the tenth iteration to be skipped.

```
1 while(i < 20) {
2     if(i == 9) {
3         i++;
4         continue;
5     }
6     ...
7     i++;
8 }
```

- **Loop: do-while+continue:** This test is a slight variation on the *Loop: while+continue* test, here a do-while construction is used instead of a normal `while` loop.

```
1 do {
2     if(i == 9) {
3         i++;
4         continue;
5     }
6     ...
7     i++;
8 } while(i < 20);
```

#### B.1.2.2 Loops with arrays

- **Loop: for, branching based on values in array:** This test is built around an array of 20 elements of which 19 are set to 0. The value of the element at index $x$ is set to 1. The elements are accessed sequentialy in a loop containing a single if statement of which the branch condition is whether the element has a value of 1. The statement(s) in the branch accessed when the value of the element is not equal to 1 use $Mem_{test}$, of which the value remains safe within the context of all statements present in the test code. The statement(s) in the other branch set the value of $Mem_{test}$ to a value that is unsafe within the context of the statements of the other branch.

```
1  for(i = 0; i < 20; i++) {
2      p[i] = 0;
3  }
4  p[x] = 1;
5
6  for(i = 0; i < 20; i++) {
7      if(p[i] == 1) {
8          ... /* Statement(s) that changes the state of the memory
               ↪ used in the other branch */
9      } else {
10         ... /* Statement(s) that is/are safe iff the other branch
               ↪ has not been executed */
11     }
12 }
```

- **Loop: for, pointer arithmetic:** This test is built around an array of 10 elements. Using pointer arithmetic all elements are accessed sequentialy in a way that could be unsafe depending on the value of the element. The number of elements that are assigned a safe value is equal to the value of parameter $x$, meaning that an $x$ with any value other than 10 will lead to a defect.

### B.1.3 Recursive loops

The tests that belong to this category are similar to the tests of the *Simple loops* category, the main difference is that recursion is used instead of `for` or `while` loops.

- **Loop: recursion:** This test contains a recursive function that is called with 20 as value for $i$. The values passed to the next iteration are the same as the one the function was called with[1], with the exception of $i$, of which the value is decreased by 1 each call.

[1]The state of $Mem_{test}$ , can change, but its location remains the same, meaning that the reference does not change.

```
int recursive_func(int i, int x, reference_to_mem_test) {
    if(i <= 1) {
        return 0;
    }

    ...

    return ... + recursive_func(i-1, x, reference_to_mem_test);
}
```

- **Loop: recursion, multiple functions:** This test is nearly identical in function the the *Loop: recursion* test. The recursive function is split in two with the first function calling the second and vice versa.

```
int recursive_func1(int i, int x, reference_to_mem_test) {
    if(i <= 1) {
        return 0;
    }

    ...

    return recursive_func2(i-1, x, reference_to_mem_test);
}

int recursive_func2(int i, int x, reference_to_mem_test) {
    if(i <= 1) {
        return 0;
    }

    ...

    return ... + recursive_func1(i, x, reference_to_mem_test);
}
```

### B.1.4 Miscellaneous

- **Goto:** This test contains a `goto` jump to an earlier part of the code, this jump is executed once. The piece of code between the goto statement and the label this statement contains an if statement of which one branch is always safe and one branch that is safe as long as $Mem_{test}$ has the value it was initialized to. The statement before the `goto` statement sets the value of $Mem_{test}$ to a dangerous value (in the context of the aforementioned branch).

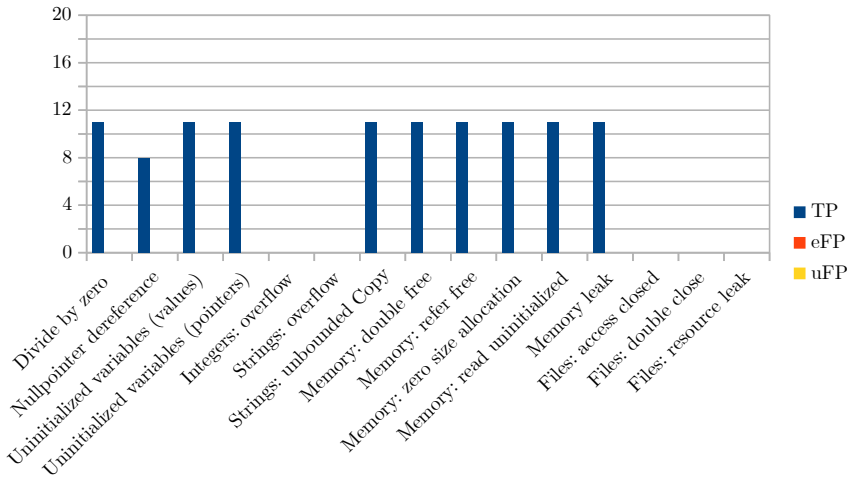
- **Struct with counter:** This test is built around a struct in which some data is stored that is used within the test. This data is accessed solely using `get` and `set` functions. Aside from this data the struct also contains an integer that functions as a counter which is initialized to `1`. This counter can be incremented

and decremented using two functions. If this counter is decremented while the counter has a value of 1 the data returned by the `get` function is no longer the same data as set by the `set` function, instead, the data returned is dangerous within the context of the program. In case of the *user after free* and *double free* defects, this test functions a little differently. Here the struct is freed upon decrementing the counter when this counter has a value of `1`, matching a simple reference counting situation. The struct is allocated using malloc at the start of the program.

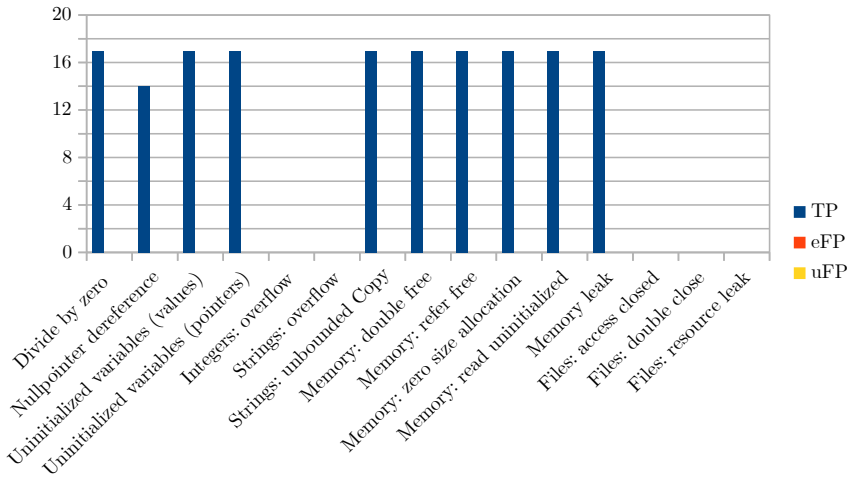
# Appendix C

# Performance numbers per defect for JM2018TS

## C.1 Tool performance per defect for JM2018TS (with entry point)

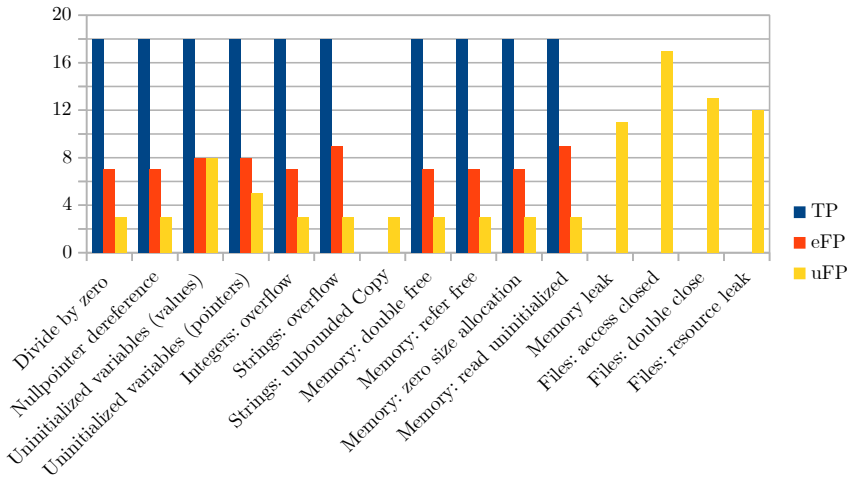



(a) Performance with the default configuration

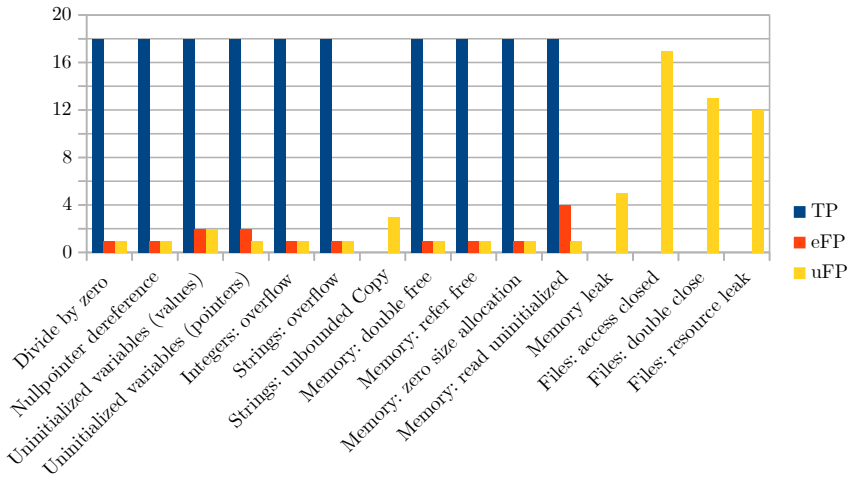



(b) Performance with improved loop analysis

Figure C.1: Performance of Clang on the different types of defects present in the new test suite

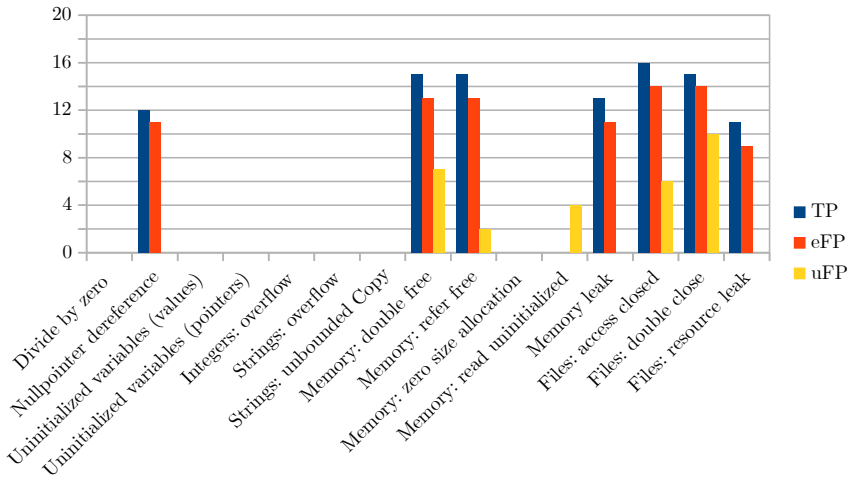



(a) Performance with the default configuration

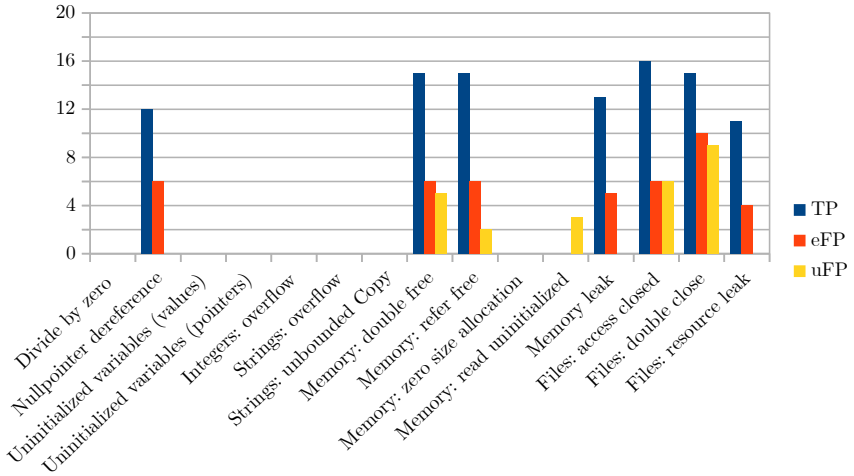



(b) Performance with improved loop analysis

Figure C.2: Performance of Frama-C on the different types of defects present in the new test suite




(a) Performance with the default configuration




(b) Performance when testing safe functions

Figure C.3: Performance of Infer on the different types of defects present in the new test suite

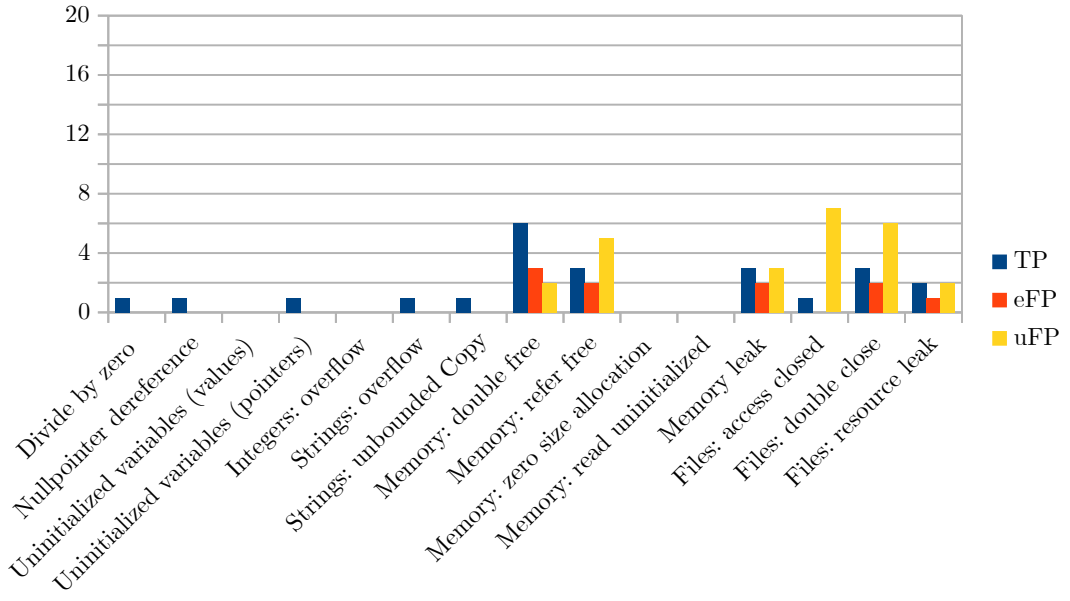
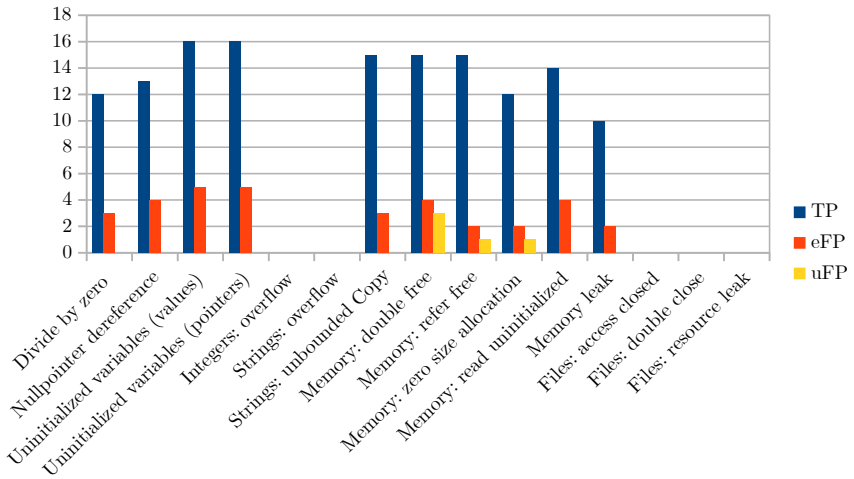
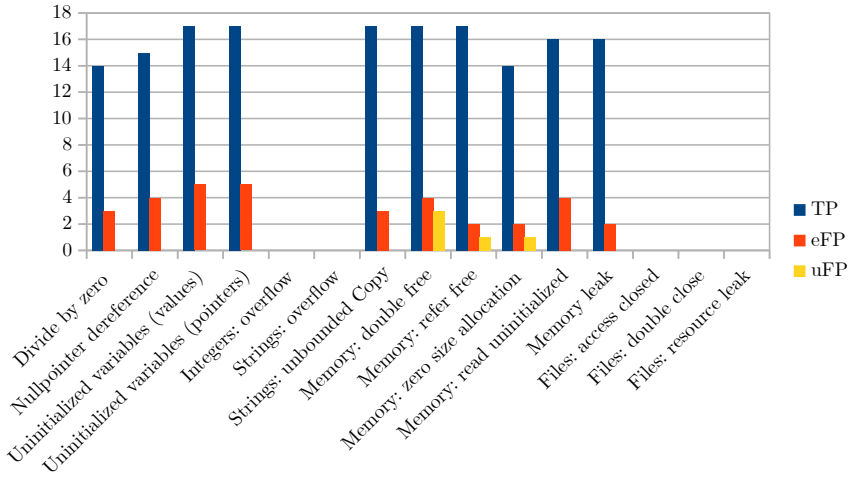



Figure C.4: Performance of cppcheck on the different types of defects present in the new test suite

## C.2 Tool performance per defect for JM2018TS, no known entry point

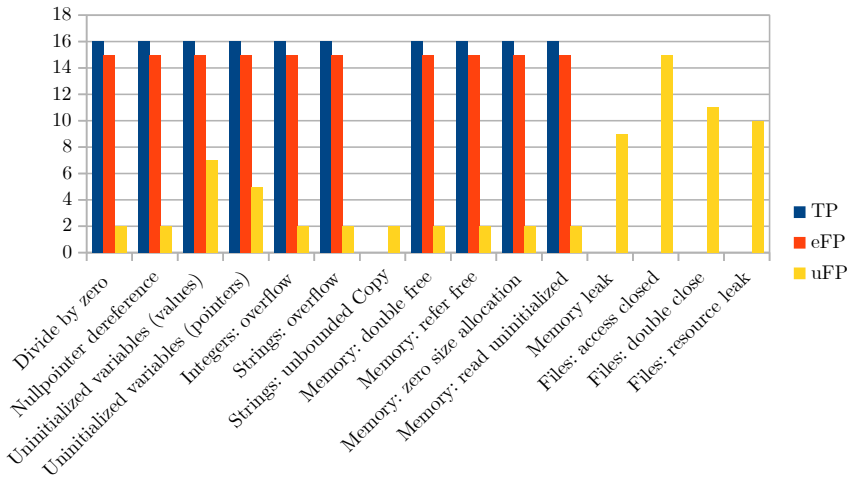



(a) Performance with the default configuration

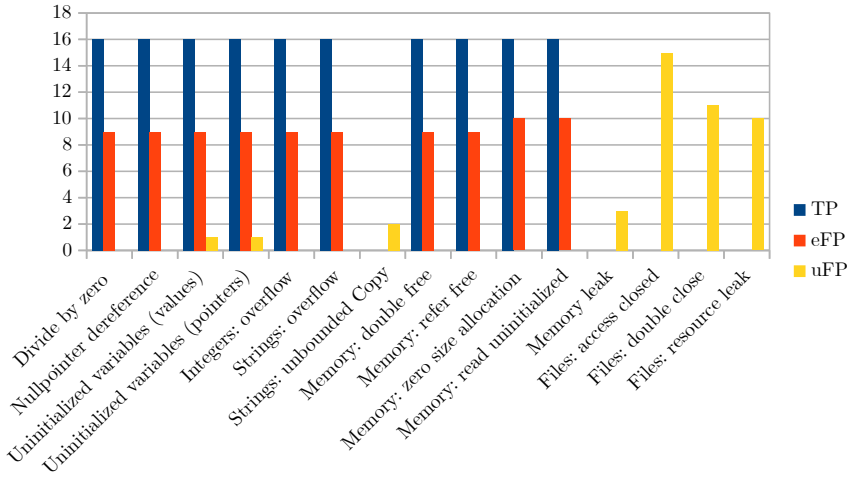



(b) Performance with improved loop analysis

Figure C.5: Performance of Clang on the different types of defects present in the new test suite




(a) Performance with the default configuration




(b) Performance with improved loop analysis

Figure C.6: Performance of Frama-C on the different types of defects present in the new test suite

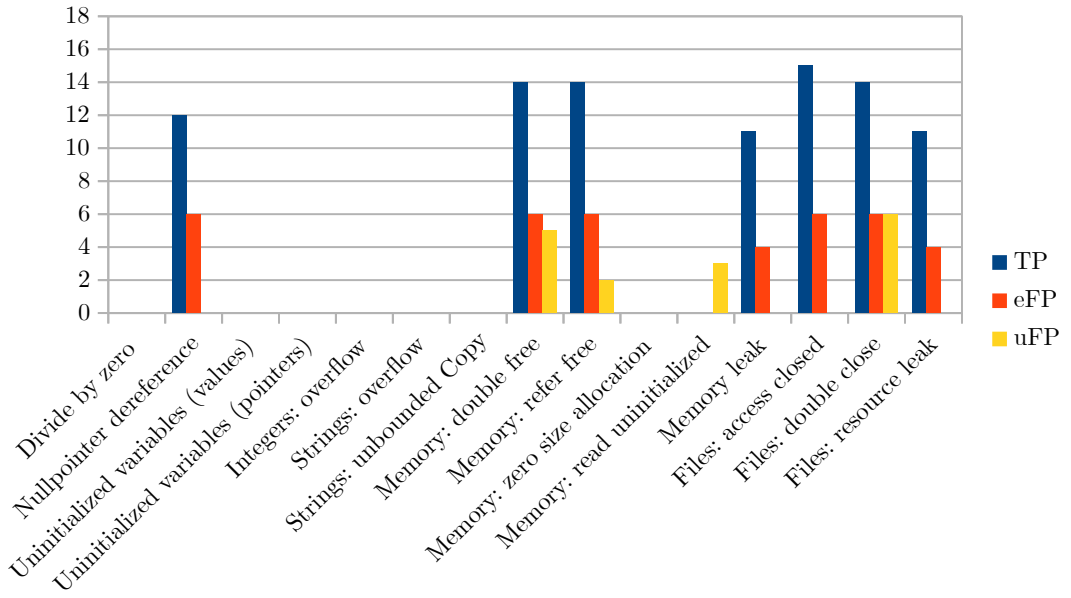



Figure C.7: Performance of Infer on the different types of defects present in the new test suite

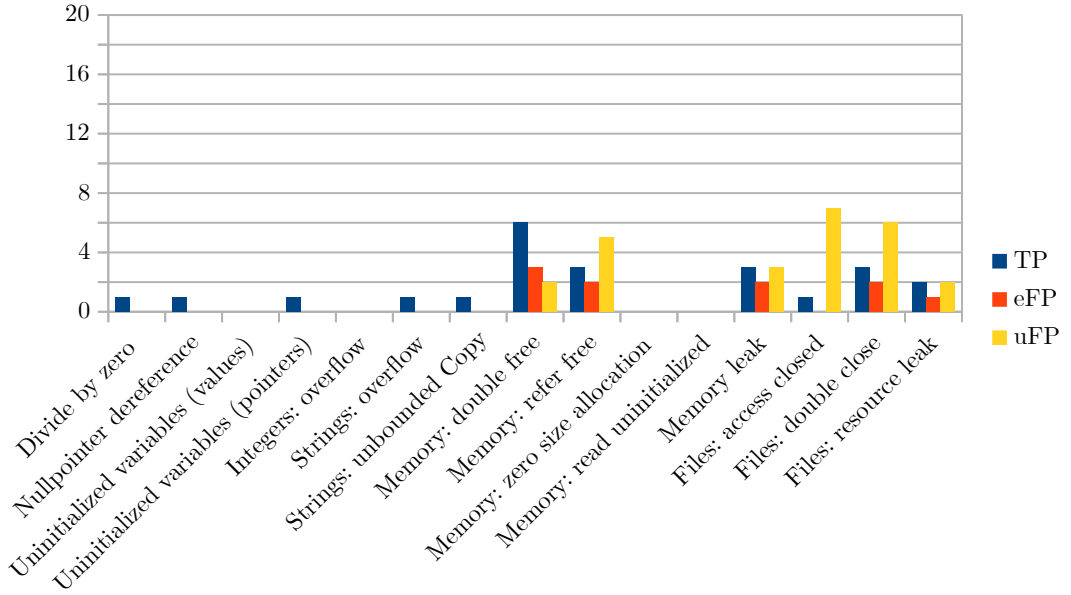



Figure C.8: Performance of cppcheck on the different types of defects present in the new test suite